On pixel-exact rendering for high-order mesh and solution

Rémi Feuillet, Matthieu Maunoury, Adrien Loseille

On pixel-exact rendering for high-order mesh and solution

Rémi Feuillet[a], Matthieu Maunoury[a], Adrien Loseille[a,*]

[a]*GAMMA Project, Inria Saclay-Île-de-France, 1 rue Honoré d'Estienne d'Orves, 91120 Palaiseau, France*

Abstract

With the increasing use of high-order methods and high-order meshes, scientific visualization software need to adapt themselves to reliably render the associated meshes and numerical solutions. In this paper, a novel approach, based on OpenGL 4 framework, enables a GPU-based rendering of high-order meshes as well as an almost pixel-exact rendering of high-order solutions. Several aspects of the OpenGL Shading Language and in particular the use of dedicated shaders (GPU programs) allows to answer this visualization challenge. Fragment shaders are used to compute the exact solution for each pixel, made possible by the transfer of degrees of freedom and shape functions to the GPU with textures. Tessellation shaders, combined with geometric error estimates, allow us to render high-order curved meshes by providing an adaptive subdivision of elements on the GPU directly. A convenient way to compute bounds for high-order solutions is described. The interest of using Bézier basis instead of Lagrange functions lies in the existence of fast and robust evaluation of polynomial functions with de Casteljau algorithm. A technique to plot highly nonlinear isolines and wire frames with a desired thickness is derived. It is based on a finite difference scheme performed on GPU. In comparison with standard techniques, we remove the use of any linear interpolation step and the need to generate *a priori* a fixed subdivided mesh. This reduces the memory footprint, improves the accuracy and the speed of the rendering. Finally, the method is illustrated with various 3D examples.

Keywords: Scientific Visualization, High-order methods, High-order meshes

1. Introduction

Numerical simulations are a convenient way to predict the behavior of physical phenomenons without using prototypes or experimentations. Several numerical simulations rely on the numerical resolution of Partial Derivative Equations (PDE). For this purpose, several steps are involved (see Fig. 1). First, a model of the studied objects is generated using a CAD (Computer-Aided Design) model [1]. Surface [2, 3] and volume [4, 5, 6, 7] meshes can be deduced from this CAD model. Using meshes, PDEs can be numerically solved with numerical schemes (FDM [8], FEM [9], FVM [10], DG-FEM [11], ...). The resolution of these PDEs gives in output a numerical solution which is an approximation of the experimental solution for the studied phenomenon.

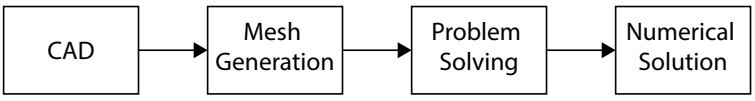


Figure 1: Main steps of a numerical simulation.

All along the process, visualization tools are mandatory to inspect the CAD model, check the conformity of the generated mesh with the model as well as its conformity with the chosen numerical method and display with high

*Corresponding author

Email addresses: remi.feuillet@inria.fr (Rémi Feuillet), matthieu.maunoury@inria.fr (Matthieu Maunoury), adrien.loseille@inria.fr (Adrien Loseille)

fidelity the numerical solution computed on the mesh. The nature of the numerical solution is related to the used numerical method. Initially tailored as piece-wise linear functions, these solutions have now evolved as piece-wise polynomial [12, 13, 14] (or even rational [15]) functions. The same evolution has also been performed with meshes. Initially meshes were composed of piece-wise linear elements, but as the order of the numerical solutions increased, the order of the elements defining the mesh has to be increased [16, 17, 18, 19, 20, 21, 22] in order to guarantee the efficiency of the associated numerical scheme [12]. However, the visualization of such meshes and solutions, also known as high-order meshes and solutions, is no that easy as most of the visualization process relies on linear features.

Increasing the degree of the mesh elements and the polynomial order of the solution enables to have more efficient methods, i.e. more accurate and with less degrees of freedom. Fig. 2 shows the approximation of an analytical function on a sphere for different degrees of meshes and solutions. Naturally, high-order elements give a better approximation of the geometry and high-order functions reduce the interpolation error.

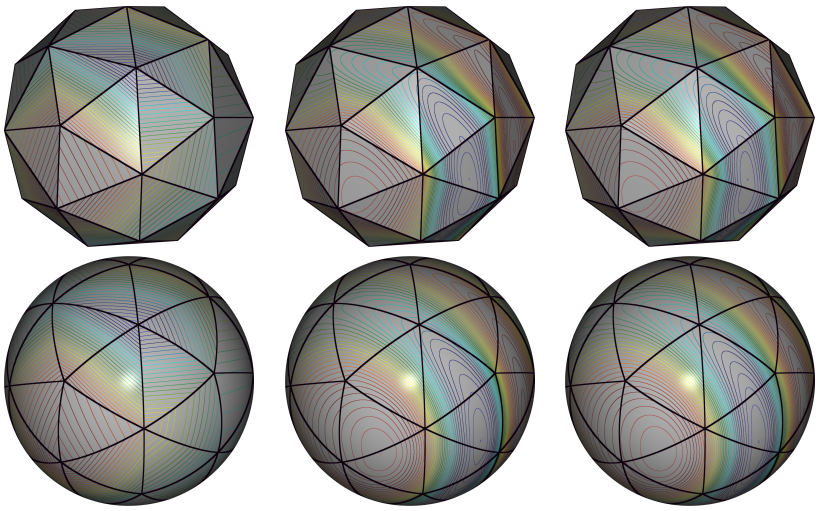


Figure 2: From left to right, P^1, P^3 and P^6 interpolations of function $cos(2\pi x)e^{(x-0.5)^2+(y-0.5)^2+z^2}$ on a P^1 (top) and P^3 (bottom) sphere.

Some works [12, 23] pointed out that without a high-order mesh, the studied physical phenomenon was not exactly solved using a high-order method. In other words, if the used mesh is not a high-order mesh, then the obtained high-order solution will never reliably represent the physical phenomenon. This issue is illustrated in Fig. 3. Euler equations are solved around a NACA0012 airfoil at the degree 5 on a linear mesh on left and on a quadratic mesh on right. The flow density is shown with its isolines. On the linear mesh, the CFD solver detects that the mesh is piecewise linear and resolves the Euler equations on the polygonal surface. This is an issue as the geometry of the NACA0012 is not piecewise linear. On the contrary, the definition of the boundary at order 2 gives a more reliable representation of the boundary of the airfoil to CFD which takes into account the fact that the geometry is curved during the numerical prediction.

In this paper, a new visualization method [24] based on features available with OpenGL 4 framework [25, 26] is proposed. This method gives a fast and light way to render high-order elements and also performs almost pixel-exact solution rendering (see Fig. 4). The main novelty of this paper is a smart use of OpenGL Shading Language to render high-order meshes and solutions. For this purpose, textures are used to transfer the degrees of freedom to the GPU, so that the solution is computed on the fly for each pixel in the fragment shader. For straight-sided elements, the high-order solution is pixel exact while it becomes almost pixel-exact for curved elements as we have to take into account the non-linear mapping of the element. Several geometric error estimates are presented to control the tessellation of the elements and we show how to use them with the tessellation shaders to render high-order meshes. A way to compute proper bounds for high-order solutions is given. Finally, we show how to display highly nonlinear isolines and wire frames with a desired thickness. Our approach is based on a GPU-based finite difference approach. This allows us to perform on-the-fly rendering, without the need to solve an ODE for isoline integration. The accuracy of

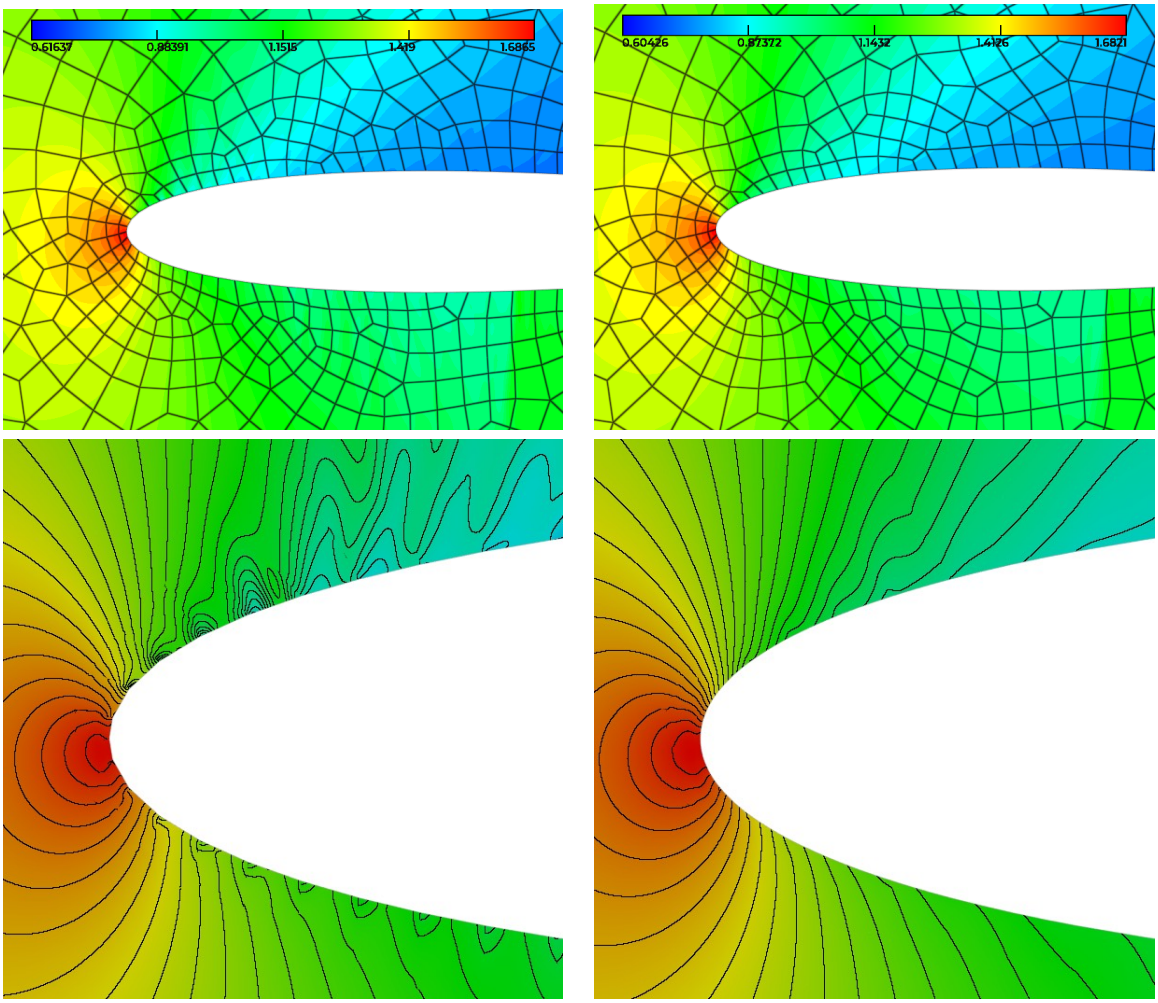


Figure 3: Degree 5 resolution of the Euler equations on a NACA0012 airfoil with a Q^1-mesh (left) and with a Q^2-mesh (right). Mesh edges are shown in black and filled iso values (top) and with darkened isolines (bottom) are displayed and show the inaccuracy in the linear representation of the geometry near the leading edge. Courtesy of Christophe Peyret (ONERA).

the process is again at the pixel-level and independent of the order of the numerical solution. All these developments have been done in the software `Vizir` 4.

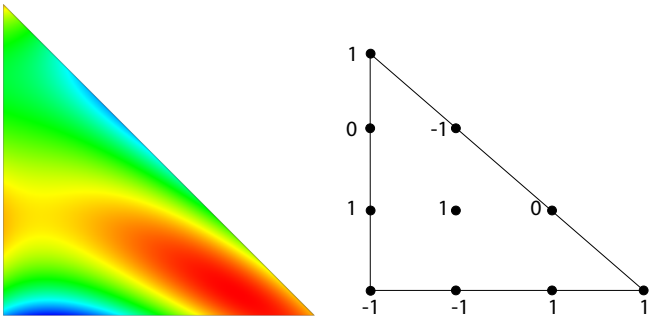


Figure 4: Left: Pixel-Exact rendering without subdivision of a P^3-solution on a triangle. Right: Values of the solution on control points.

The paper is outlined as follows. Section 2 reminds what a high-order mesh and a high-order solution are. Section 3 presents the existing methods which tackle our visualization problem. Section 4 gives an overview of the used technology. Section 5 deals with the visualization of high-order meshes and Section 6 deals with the high-order solution rendering. Examples are given all along the paper and more complex examples are described in Section 7.

2. Preliminaries

2.1. High-order mesh

In general, a finite element is defined [27] by a triplet $\{K, \Sigma_K, V_h\}$ where K denotes the geometric element (triangle, etc), Σ_K the list of nodes of K, and V_h the space of the *shape functions*, here it will be the Lagrange polynomial functions (or interpolants). To properly define the geometry and these functions, a reference space (that can also be seen as a parameter space) is defined where all coordinates are, in general, between 0 and 1. In this space, the reference element is denoted $\widehat{K}$ and has a fixed (and sometimes uniform) distribution of nodes. The element K, also called physical element, is thus the image of $\widehat{K}$ via a mapping, denoted F_K (see Fig. 5). More specifically, for each point M of K, there is a point $\widehat{M}$ of $\widehat{K}$ such that $M = F_K(\widehat{M})$. In particular, the position of a point M inside K is defined by

$$M = \sum_{i=1}^{n} \phi_i^n(\widehat{M}) A_i, \tag{1}$$

where n is the number of nodes, $A_i = F_K(\widehat{A_i})$ with $\widehat{A_i}$ the nodes of the reference element which map to A_i the nodes of the physical element, and ϕ_i^n are the Lagrange polynomial functions defined such that:

$$\phi_i^n(\widehat{A_j}) = \delta_{ij} \quad \text{and} \quad \sum_{i=1}^{n} \phi_i^n = 1.$$

Now, we will assume that the shape functions ϕ_i^n are polynomial functions of degree d of the parameters space and therefore define high-order finite elements. In this case, the number of nodes n is a consequence of the degree of the element. A convenient way to handle high-order meshes is to write F_K into Bézier form [27, 28] using Bernstein polynomials B_{ijk}^d, like for a triangle:

$$M = \sum_{i+j+k=d} B_{ijk}^d(u, v, w) P_{ijk}, \tag{2}$$

with $B_{ijk}^d(u, v, w) = \frac{d!}{i!\,j!\,k!} u^i v^j w^k$ and (u, v, w) the (barycentric) coordinates of the parameters space. The points $(P_{ijk})_{i+j+k=d}$, are the Bézier control points and are directly related to the nodes $(A_i)_{1 \le i \le n}$. The same kind of equivalence can be made for the other elements.

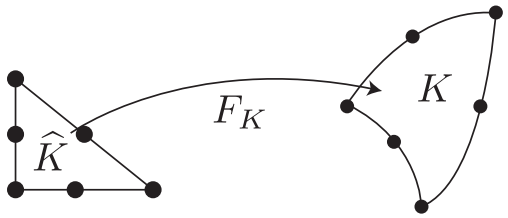


Figure 5: Mapping F_K from $\widehat{K}$ to K.

2.2. High-order solution

A high-order solution defined on a finite element mesh can be written in the same manner as the high-order mesh. For every element of the mesh, a solution of degree d' (where d' is not necessarily the degree d of the element) is defined on the reference element $\widehat{K}$ (the same as the geometrical elements) using Lagrange shape functions:

$$f = \sum_{i=1}^{n'} \phi_i^{n'}(\widehat{M}) f_i, \tag{3}$$

where f_i is the value of the solution at a set of nodes $\widehat{A}_i'$ of the reference element. Note that equation (3) is very similar to equation (1). Likewise, the solution can be written into Bézier form.

When $d = d'$, we speak of isoparameterization. An important remark is that the correspondence of f to the coordinates of the physical space is only polynomial when the element is linear (e.g. when the parameters coordinates are linearly related to the physical coordinates). In other cases, it is not true. In particular, a quadrilateral of degree 1×1 defines a bilinear mapping which is not a linear mapping when the geometrical element is not a rectangle. In this case, the solution defined on the element is not polynomial in space.

Using the Bézier form to represent high-order curved meshes or solutions is particularly convenient as it naturally conveys geometric information and offers robust, fast and reliable algorithms for computing bounds ord to evaluate curvature or the non-linearity of the underlying mapping.

3. State of the art

Classic visualization software like `ParaView` [29], `TecPlot` [30], `FieldView` [31], `Ensight` [32], `Medit` [33], `Vizir` (OpenGL legacy based version) [34], `Gmsh` [35], ... historically rely on the display of linear triangles with linear solutions on it. More precisely, each element of the mesh is divided into a set of elementary triangles. At each vertex of the elementary triangle is attached a value and an associated color. The value and the color inside the triangle is then deduced by a linear interpolation inside the triangle. With the increase of high-order methods and high-order meshes, these software adapted their technology by using subdivision methods. If a mesh has high-order elements, these elements are subdivided into a set of linear triangles in order to approximate the shape of the high-order element [36]. Likewise, if a mesh has a high-order solution on it, each element is subdivided into smaller linear triangles in order to approximate the rendering of the high-order solution on it. The subdivision process can be really expensive if it is done in a naive way (uniform tessellation). For this reason, mesh adaptation procedures [37, 38, 39, 40] are used (see Fig. 6) to efficiently render high-order solutions and high-order elements using the standard linear rendering approaches. However, a visualization error, due to the approximation of a high order function by affine functions, is introduced and the rendering obtained is, as a consequence, inaccurate (see Fig. 6). Furthermore, the use of a non-conforming mesh adaptation can lead to a discontinuous rendering for a continuous solution (see Fig. 6).

In Fig. 7, several rendering of a Q^1-solution on a Q^1-quadrilateral with different software are compared. As the solution is not linear, a visualization error is introduced by classic software. Hence, `Medit`, `Vizir legacy` and `ParaView` generated a subdivision of two triangles while a tessellation of 8×8 quadrilaterals is done in `Gmsh`, and it leads an inaccurate representation of the solution and its isolines. In comparison, thanks to the pixel exact representation, the non-linearity of the solution is perfectly rendered with our approach.

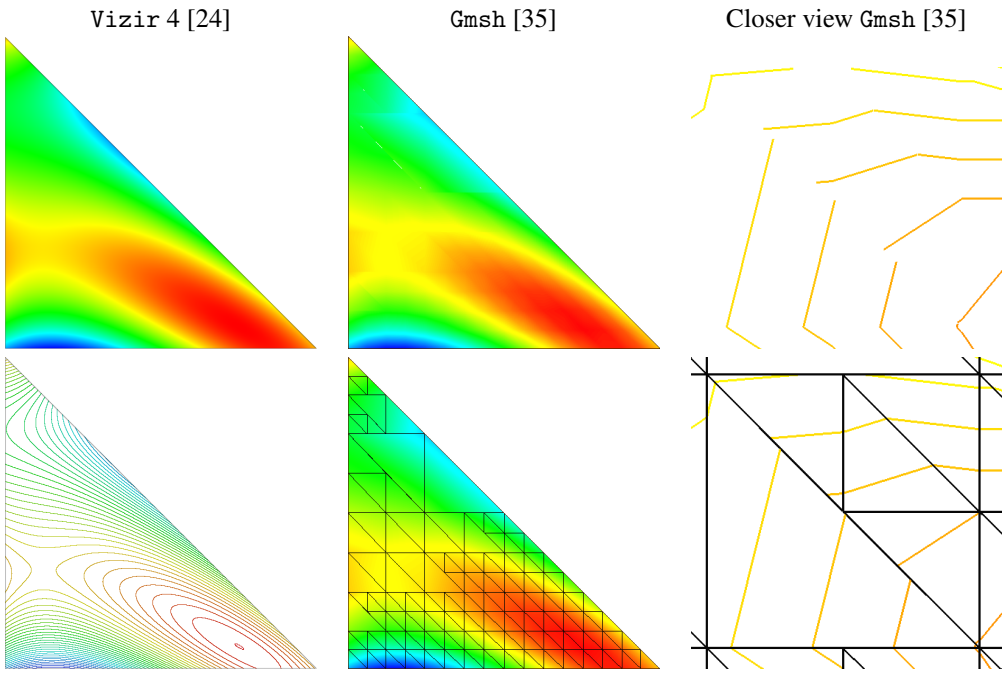


Figure 6: Rendering of a P^3-solution on a triangle. Left: Our pixel-exact representation (top) and its iso values (bottom). Middle: Representation (top) of the solution linearly subdivided on an adaptive grid generated by Gmsh [35], version 4.2.1, with 1% of visualization error, tessellation of 169 sub-triangles (bottom). Right: Zoom on isolines near the barycenter of the triangle for the approximation with 169 sub-triangles. Discontinuities are shown (top) and due to the non-conforming tessellation (bottom).

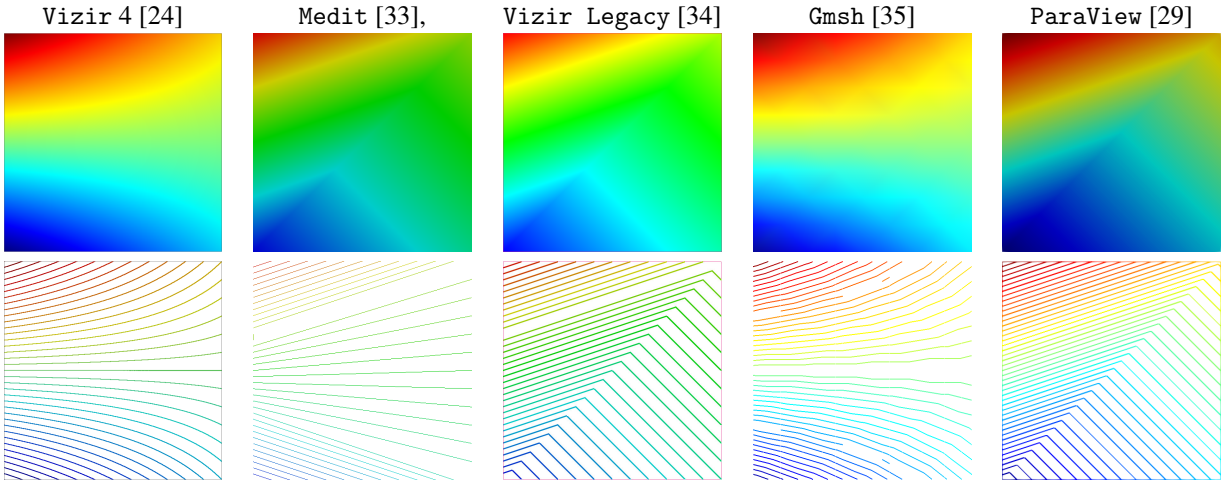


Figure 7: Rendering of a Q^1-solution on a Q^1-quadrilateral with different software: representation (top) and isolines (bottom). A subdivision of 2 triangles is done in `Medit`, `Vizir legacy` and `ParaView`, while a tessellation of 8×8 quadrilaterals is done in `Gmsh`. No tessellation is done in `Vizir 4` (our method) as a pixel exact rendering is performed.

Even when optimized these approaches have a huge RAM memory footprint as the subdivision is done on CPU in a preprocessing step. Also the adaptive subdivision process can be dependent on the palette (e.g. the range of values where the solution is studied) as the color only varies when the associated value is in this range. In this case, a change of palette inevitably imposes a new adaptation process.

Other approaches are specifically devoted to high-order solutions and are based on ray casting [41, 42, 43]. The idea is for a given pixel, to find exactly its color. To do so, for each pixel, rays are cast from the position of the screen in the physical space and their intersection with the scene determines the color for the pixel. If high-order features are taken into account, it determines exactly the color for this pixel. However, this method is based on two non-linear problems: the root-finding problem and the inversion of the geometrical mapping. These problems are really costly and do not compete with the interactivity of the standard linear rendering methods even when these are called with a subdivision process.

The proposed method intends to be a good compromise between both methods. It guarantees pixel-exact rendering on linear elements without extra subdivision or ray casting and it keeps the interactivity of a classic method. Moreover, the subdivision of the curved entities is done on the fly on GPU which leaves the RAM memory footprint at the size of the loaded mesh and solution.

4. Presentation of the OpenGL 4 graphic pipeline

In this section, we present the OpenGL 4 rendering pipeline. More details on OpenGL 4 and in particular OpenGL Shading Language (GLSL) can be found in [25, 26]. We emphasize how we use the flexibility for high-order element and solution rendering. The customizable part of the OpenGL 4 graphic-fixed pipeline is based on the use of shaders. They are GLSL source code files that replace parts of the OpenGL pipeline. The main pros of redefining all the shader stages are:

- the memory footprint in RAM is limited to the size of the mesh,
- addition (subdivided) entities are created on the graphic cards directly on the fly,
- solutions are computed in the fragment shader so that new shape functions and interpolation schemes can be interchanged.

The OpenGL 4 pipeline can be customized with up to five different shader stages (see Fig. 8). In general, a shader receives its input via programmer-defined input variables, and the data for those variables come either from the main OpenGL application or previous pipeline stages (other stages). Data can also be provided to any shader using uniform variables or textures [25].

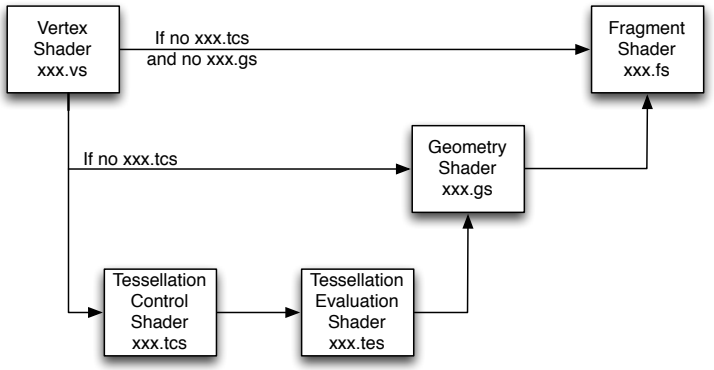


Figure 8: Shaders used for the OpenGL graphic pipeline.

More precisely, the description of the shaders is the following:

- The **Vertex Shader** (VS). Its input variables are vertex attributes. The data corresponding to the vertex position must be transformed into clip coordinates and assigned to the output variable `gl_Position`. This shader can also send other information down the pipeline using shader output variables.

- The **Fragment Shader** (FS). Its input variables come from the graphic pipeline and is a transformation of other shader outputs. The fragment shader determines the appropriate color for the pixel and sends it to the frame buffer using output variables. Also, among built-in variables, we can pass through our own variables which means that for a pixel, we can deduce (x, y, z), (u, v), or primitive ids. Using these variables, it is possible to display anything which is a function of these variables. In Fig. 9), we illustrate the pixel-exact rendering of the non linear function that is computed on the fly. In our context, the Fragment Shader is also used to perform the wire frame rendering as well as high-order solution and isoline rendering (see Section 6). For the storage of raw data (like high-order solutions), textures are used.

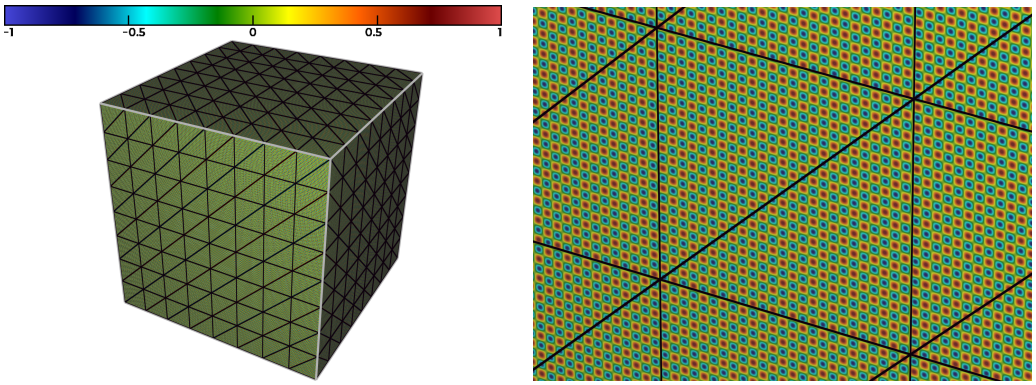


Figure 9: Example of an exact rendering of the function $sin(100\pi x) + sin(100\pi y) + sin(100\pi z)$ on a mesh of the unit cube. This is performed by using the Fragment Shader.

These two shaders can be enough to define a customization of the graphic pipeline. In this case, between the two shaders, the vertices are assembled into primitives, clipping takes place and the viewport transformation is applied.

However, the OpenGL pipeline can be even more customized thanks to the following shaders:

- The **Geometry Shader** (GS). This shader cannot be used without the two previous shaders. It has access to all the input data of the vertices of the primitive that can be provided either by the vertex shader or by the tessellation evaluation shader. The GS can receive some primitives and can emit other primitives as long as only one type of primitive is in input or output. For instance, it can receive vertices and output triangles. The GS functionality is centered around two primitives: `EmitVertex` and `EndPrimitive`. To each vertex of the primitive, all useful data are linked with it and then the vertex is emitted thanks to the first primitive. Once all vertices of the primitive are processed, the second primitive is called. Also, every data attached to each vertex is by default linearly interpolated inside the primitive and sent to the fragment shader. In our context, the GS can also be used to define the flat shading (see Fig. 10, left) and the shrinking for the linear elements (see Fig. 10, middle). For all the elements, when a clipping plane is used, the operation is performed here (see Fig. 10, right) and finally, the GS can be used to propagate the physical and parametric coordinates (x, y, z) and (u, v) as well as the normals. More details are given in Section 5.

- The **Tessellation Control Shader** (TCS) and the **Tessellation Evaluation Shader** (TES). These shaders cannot be used without the three previous shaders. As soon as tessellation shaders are used, the only used primitives are patches. A patch primitive is a part of the geometry defined by the programmer. The number of vertices by patch is configurable as well as the interpretation of the geometry. For instance, the patch can be used as a set of control points that defined an interpolated curve or surface (Bézier curve for instance). More precisely, the TCS sets up the Tessellation Primitive Generator (TPG) by defining how the primitives should be generated by it and in particular in how many sub-entities the element should be divided (e.g. tessellated). In our context,

the level of discretization (e.g. the granularity of the tessellation) given by the TCS is controlled with basic geometrical error estimates so that straight elements are not subdivided more than necessary. More details are given in Section 5.2. Once the discretization is done, the entities are sent to the TES. For a given vertex, the TES determines the position of the vertices in the physical space thanks to the coordinates of the parameter space and also other vertex-related data. All the data are then sent to the GS that process it like any other entity. In our case, these shaders are used to approximate high-order curved elements such as P^k-triangles/edges ($k \geq 2$) and Q^k-quadrilaterals ($k \geq 1$).

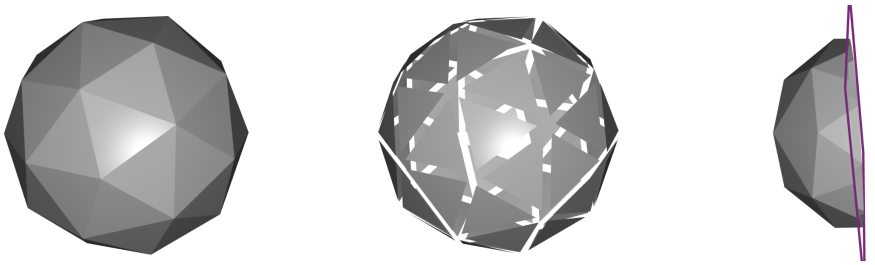


Figure 10: Example of three different operations performed with the Geometry Shader: flat shading, shrinking and clipping.

5. High-order elements visualization

5.1. Visualization of surface elements

For simple shapes like P^1-triangles and edges, no tessellation is needed as these elements are planar. In this case, only three shaders are used: the Vertex Shader, the Geometry Shader (which outputs triangles or lines) and the Fragment Shader. For more complex and potentially non-planar geometries (Q^1-quadrilateral, or any higher-order element), the two tessellation shaders are added to the three others. To process any geometry, it is first put into its Bézier form and then the obtained control points are sent to the graphic pipeline. Control points are preferred to Lagrange points as their use minimizes the number of operations which is a great property when it comes to GPU programming. In output, a reliable representation of the geometry is displayed on the screen.

For the sake of clarity, let us have a look on the case of a P^3-triangle we would like to display (see Fig. 11). A P^3 triangle is generally defined by its 10 Lagrange points (here in Bézier notations): $(M_{ijk})_{i+j+k=3}$ (see Fig. 11, left).

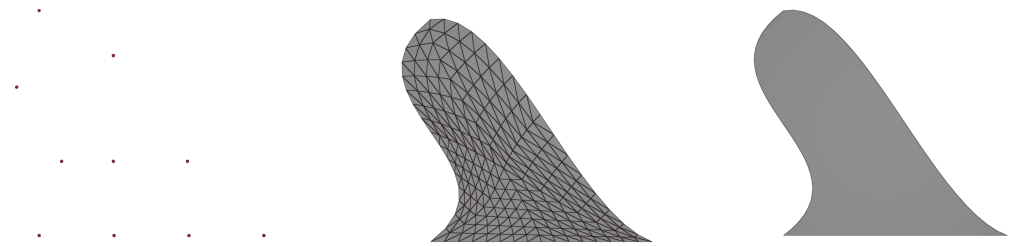


Figure 11: Different steps involved in the drawing of P^3-Triangle: the control points, the tessellation and the rendering.

The first step is to compute the control points. It is done in a hierarchical fashion: the points lying on the edges of

the surface first and the points lying inside the surface. For the first edge, we have:

$$\begin{cases} P_{300} = M_{300}, \\ P_{120} = \dfrac{18M_{210} - 9M_{120} - 5M_{300} + 2M_{030}}{6}, \\ P_{210} = \dfrac{18M_{120} - 9M_{210} - 5M_{030} + 2M_{300}}{6}, \\ P_{030} = M_{030}, \end{cases}$$

and the same goes for the two others. The inner surface control point can be deduced:

$$P_{111} = \frac{9M_{111}}{2} - \frac{P_{300} + P_{030} + P_{003}}{6} - \frac{\sum_{j=1}^{2} \sum_{i=1}^{2-j} P_{ij0} + P_{i0j} + P_{0ij}}{2}.$$

For every P^3 triangle, its 10 control points are sent to the graphic pipeline. The first step is the processing by the Vertex Shader. In this shader, the coordinates of each vertex are scaled and multiplied by the viewport and model matrices so that they are transformed into clip coordinates. As the element is curved, the next shader is the Tessellation Control Shader (for linear elements, it goes directly to the Geometry shader). In this shader, the number of subdivisions along each edge of the geometry and the number of subdivisions inside the surface are set. Thanks to these parameters, a tessellation (e.g. a subdivision in triangles or lines) of the parameters space is generated by the Tessellation Primitive Generator. The Tessellation Evaluation Shader is then executed for each point of the tessellation. In the case of a triangular element, these points are characterized by a triplet (u, v, w) with $0 \leq u, v, w \leq 1$ and $u + v + w = 1$ which is the corresponding sampling in the parameters space. Thanks to this triplet and following the definition given in equation (2), the coordinates of the corresponding point in the physical space are computed:

$$M(u, v, w) = \sum_{i+j+k=3} \frac{3!}{i!j!k!} u^i v^j w^k P_{ijk}. \tag{4}$$

In a similar way, the geometric normal are computed. This way, an approximation made of triangles of the curved geometry is performed on the fly on the GPU. Each sub-triangular element is then sent to the Geometry Shader. In this shader, all useful data related to each vertex defining the (sub) elements are attached: parametric coordinates, physical coordinates, distance to the clipping plane (if any). Once all these data are set, they are linearly interpolated to define them inside the primitive and sent to the Fragment Shader. The Fragment Shader takes care of what happens for each fragment/pixel related to the primitive previously processed. It gives a wanted background color to the pixel and applies a shading to it (toon, diffuse or Phong model) using the normals provided by the previous shader (see Fig. 12). It is also able to perform wire frame, numerical solution and isoline rendering. This will be explained in Section 6.3.

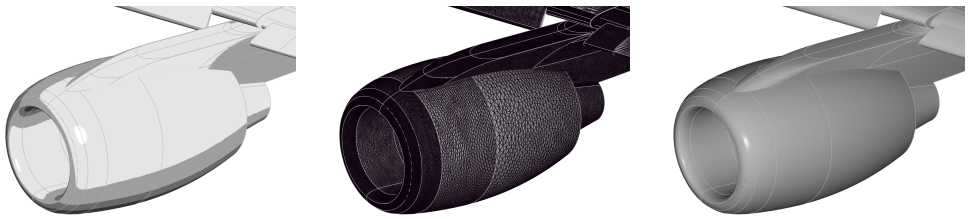


Figure 12: Example of three different displays (toon, wire and Phong model) on an engine of a Dassault Falcon.

By performing all these previous steps, it is possible to display any type of surface element. It can be noted that the display of the tessellation grid (like in Fig. 11 middle) can detect if a high-order element is valid or not. Indeed, if the grid is not properly mapped because sub-triangles overlap themselves, this means that the mapping is not invertible as the physical position of the points of the tessellation grid is set by the mapping.

5.2. Control of the granularity of the tessellation

In the case of curved elements a tessellation may help to properly display the geometric features but it is not always mandatory. First, let us have a closer look at what happens in the Tessellation Control Shader (see Fig. 13). The subdivision of an element is controlled through two types of integer variables `TessLevelOuter` and `TessLevelInner`. `TessLevelOuter[i]` is the number of subdivisions along the $i-th$ line of the element, and `TessLevelInner` is related to the sub element inside (if any) and sets the number of subdivisions for the second layer of triangles along the axis with the associated edge. An uniform discretization is then set in the inner element.

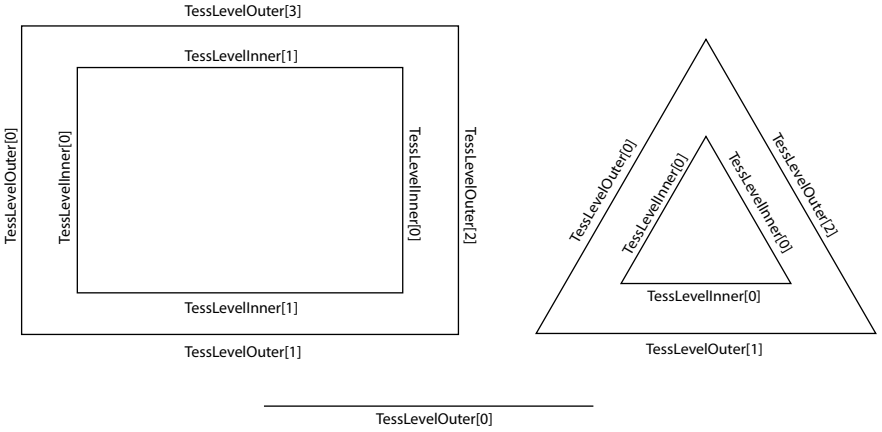


Figure 13: Variables controlling the subdivision for each type of element in the Tessellation Control Shader. Top: Quadrilateral (left) and Triangle (right). Bottom: Edge.

The idea is then to set up simple but fast error estimates on the geometrical approximation of a high-order element by its equivalent planar element. Based on these error estimates, the level of tessellation of these elements can be set. In order to have a conforming tessellation of the global mesh, the discretization of the lines is done independently of what happens inside the element.

Error estimate for an edge or a line. On a line or an edge, if a control point lying on an edge is denoted P_i^{edge} and its equivalent control point on the straight edge is denoted $P_i^{straight}$, the following point-wise error estimate ϵ is considered:

$$\epsilon(P_i^{edge}) = \frac{\|P_i^{edge} - P_i^{straight}\|}{l_{edge}}, \tag{5}$$

where l_{edge} is the length of the edge defined by the extremities. If the extremities are the same (case of a looping edge), l_{edge} is replaced by the length of the polygonal curve defined by the control points of the edge. This error estimate is computed for every control point lying on a given edge. The error estimate ϵ_{edge} associated to the edge is then the largest of its control points error estimates:

$$\epsilon_{edge} = \max_i \epsilon(P_i^{edge}). \tag{6}$$

The associated `TessLevelOuter` can then be set using the following formula:

$$\texttt{TessLevelOuter} = 1 + [5t\epsilon_{edge}],$$

where t is a user integer parameter defining the level of discretization of the line. In practice, t is a *uniform* (in the sense of GLSL) variable given to the shaders which can be modified on the fly. This means that a line is subdivided $n+1$ times if

$$n.l_{edge} \leq 5t.\max_i(\|P_i^{edge} - P_i^{straight}\|) < (n+1)l_{edge}.$$

In particular, if the element is straight, the number of subdivisions is always equal to 1. For instance in the P^3-triangle of Fig. 14, the bottom edge is not divided as it is a straight edge, and the right edge is less divided than the left edge because the first one is closer to the straight edge than the second one.

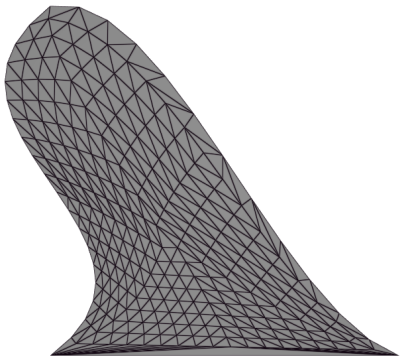


Figure 14: P^3-Triangle of Fig. 11 discretized with an adaptive tessellation.

Error estimate inside elements. For the number of subdivisions inside the face, the idea is more or less the same: if a control point lying on the inner part of a face (if any) is denoted P_i^{face} and its equivalent control point on the straight face is denoted $P_i^{straight}$, the following point-wise error estimate ϵ is considered:

$$\epsilon(P_i^{face}) = \frac{||P_i^{face} - P_i^{straight}||}{l_{edge}^{max}},$$

where l_{edge}^{max} is the largest edge length of the straight element. The error estimate ϵ_{face} associated to the inner part of face is then the largest of its control coefficients error estimates:

$$\epsilon_{face} = \max_i \epsilon(P_i^{face}). \tag{7}$$

Error estimate for quadrilaterals. When dealing with quadrilaterals of degree d, another error estimate ϵ_{quad} is considered:

$$\epsilon_{quad} = \frac{|\det(P_{0d} - P_{00}, P_{dd} - P_{00}, P_{d0} - P_{00})|}{||P_{0d} - P_{00}||\,||P_{dd} - P_{00}||\,||P_{d0} - P_{00}||}. \tag{8}$$

Indeed, this error estimate has the ability to detect if a quadrilateral is non-planar which can be true even if $d = 1$. Fig. 15 shows that if a non-planar quadrilateral is not tessellated, its true shape is not truly represented by its decomposition into two triangles.

Tessellation inside elements. Based on these estimators (6), (7) and (8), the `TessLevelInner` is then set:

$$\texttt{TessLevelInner} = 1 + [5t \max(\epsilon_{edge}, \epsilon_{face}, \epsilon_{quad})], \tag{9}$$

using the same t as with `TessLevelOuter`. Note that if the elements are straight, all the ϵ are equal to 0 and consequently `TessLevelOuter` = `TessLevelInner` = 1. A triangle and an edge are then not divided and a quadrilateral is divided into two triangles, which is the classic way to visualize these straight elements in OpenGL Legacy.

5.3. Volume elements visualization

The rendering of the volume elements is done in an indirect way. Indeed, only a surface deduced from the elements is rendered. The idea is to render the external surface of the elements. This surface can either be the external part of a volume provided without surface or the external surface of elements intersected by a cut plane. In the last case, the display of the cut plane is carried out with clipping so that the external surface is half-visible in the same time (see Fig. 16). In this case, the rendering process is exactly the same as a surface rendering with quadrilaterals, triangles and edges.

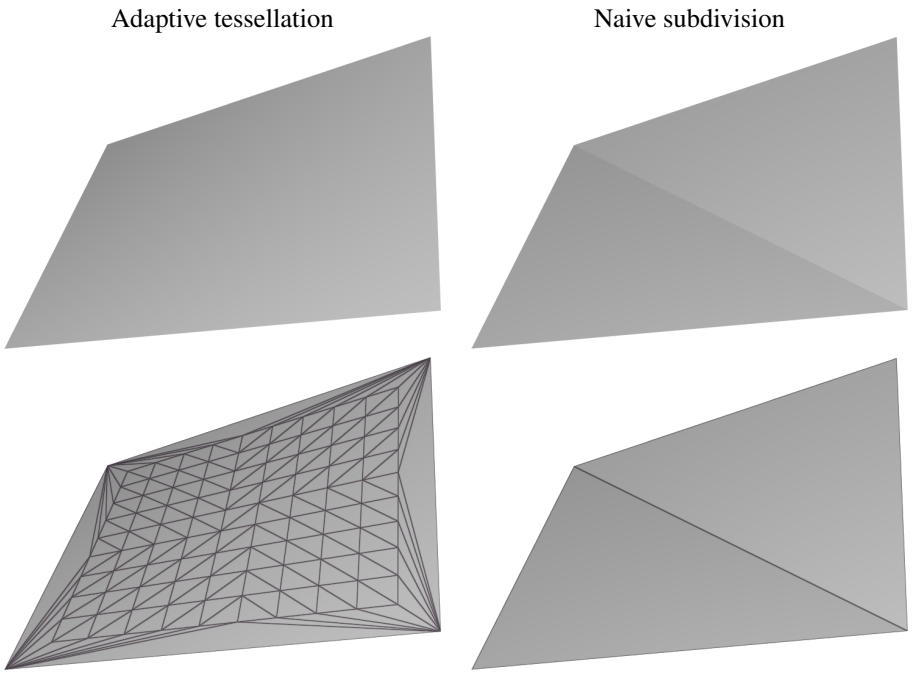


Figure 15: Various renderings of a non-planar Q^1-quadrilateral. Left: rendering with a tessellation (our method). Right: rendering with OpenGL Legacy. Bottom: Used tessellation.

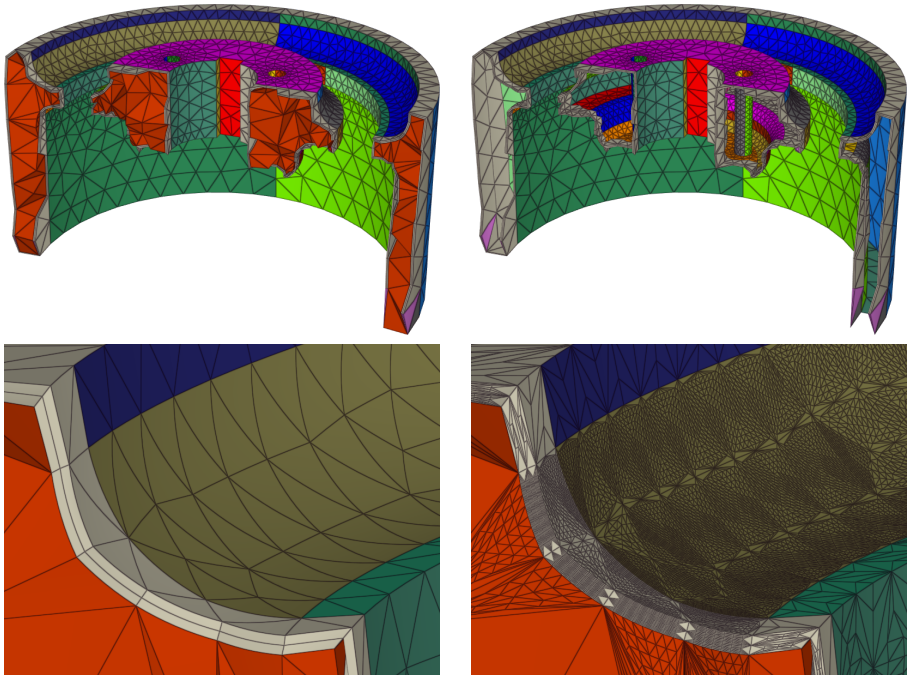


Figure 16: Rendering of a mesh composed of hybrids P^2-elements with clipping. References (patch ids) are shown in color. Top Left: clipping is activated (only a part of the mesh is shown). Top Right: The volume (in red in this figure) is hidden. Bottom Left: Zoom on some curved elements. Bottom Right: The tessellation of these elements is displayed. Courtesy of Loïc Maréchal (Inria).

6. High-order solutions visualization

6.1. Almost pixel-exact solution rendering

The OpenGL graphic pipeline also offers the possibility to display high-order (scalar) solutions. To do so, the Fragment Shader is used. As seen in the previous section, when considered, the Fragment shader is attached to a pixel representing a fragment of a given primitive associated to an element. In particular, it receives the interpolation of the various data attached to each one of the vertices defining the primitive. It interpolates normals, parametric and physical coordinates. Thus, if a given element is defined by a linear mapping, it has access to the exact set of physical coordinates associated to a given set of parametric coordinates. When the mapping is not linear, it gives an approximation, depending on the level of the tessellation, of the set of physical coordinates associated to a given set of parametric coordinates. Since the parametric coordinates are available in the fragment shader, the only thing left to display high-order solutions is to provide the value of the solution at each degree of freedom of each element. A convenient way to transmit these data is to use textures that can store raw data of large size. In practice, it is the control solutions (e.g. the coefficients of the solution in the Bernstein basis) that are stored in the texture as defined in Section 2.

Once parametric coordinates and control solutions are obtained, the solution can be computed using its exact polynomial expression. The effective computation of this solution is based on a de Casteljau's algorithms [27, 28] which can evaluate with a minimal number of operations a polynomial of any degree using its control coefficients. The value is then known for each fragment. The main consequence of all this is that if the mapping is linear then the computed solution is pixel-exact, otherwise, it is almost pixel-exact.

Furthermore, a special treatment is performed for the display of a solution at planar quadrilaterals. Indeed, the subdivision in 2 triangles is enough for the visualization of the shape of the quadrilateral as the two triangles span the same shape as the considered quadrilateral. However, it is not enough for the rendering of a solution on it (see Fig. 17). In fact, by using the Geometry Shader, the parametric coordinates are interpolated inside each of the sub-triangle in a linear fashion, but not in a bilinear fashion. Consequently, if the quadrilateral is not a parallelogram, the solution will not be reliably rendered. For this reason, another error estimate is considered:

$$\epsilon^{sol}_{quad} = \frac{\|P_{00} - P_{d0} - P_{0d} + P_{dd}\|}{\max(\|P_{0d} - P_{00}\|, \|P_{dd} - P_{0d}\|}. \tag{10}$$

This error estimate has the ability to detect if the mapping associated to a Q^1 quadrilateral has quadratic terms or not. When a solution is displayed on a quad, ϵ_{quad} is replaced by $\max(\epsilon_{quad}, \epsilon^{sol}_{quad})$ in (9). Note this error estimate is not used when the mesh is displayed without solution rendering.

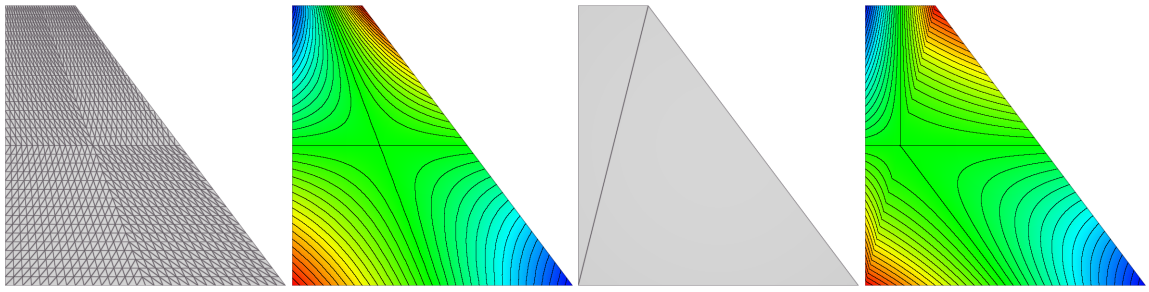


Figure 17: Rendering of a Q^1-solution on a non-linear Q^1-quadrilateral with two error estimates ϵ^{sol}_{quad} defined by (8) shown in the two first pictures and ϵ_{quad} defined by (10) shown in the two last pictures. From left to right: a fine tessellation has been created by ϵ^{sol}_{quad}; it leads to a good rendering and isolines (in black); ϵ_{quad} gives a decomposition in only two triangles as it is co-planar; it leads to a bad solution and false isolines.

Finally, to display a numerical solution, it is mandatory to have a palette and its color map (e.g. a range of values for the solution and its associated color in RGB). In a same manner, these two arrays are sent to the Fragment Shader using textures. The next step is therefore to transform our computed solution into a RGB vector. First, we check the

values of the bounds of the palette and if the solution is not inside the bounds, it is set to the bound of closest value. The RGB vector is then deduced using the color map array.

6.2. Computation of proper bounds for a high-order solution

When displaying a numerical (scalar) solution, a palette is always used to determine the color associated to a solution value. It can be set by the user, but in some cases, the user is not aware in advance of the variations of its numerical solution and in particular of its range of values. Classic visualization methods use the sampling of the solution on the visualization grid to get it. However in our case, the solution evaluation is performed in the Fragment Shader and no easy callback of the highest and lowest computed value is possible.

The objective is therefore to find proper bounds for a high-order solution (in a preprocessing step) and then to use them as a default palette. If the solution is linear (or constant), it is straightforward as the extrema on each element lies on the vertices. In this case, the bounds of the solution are the largest and the smallest values of the solution at the vertices. For other polynomial solutions, it is not necessarily the case and a refinement procedure [18, 20, 27, 44] is required.

The idea is to subdivide elements, evaluate the solution in these sub-elements with a de Casteljau algorithm and update the bounds. The complete algorithm is given in Algorithm 1.

Algorithm 1 Computation of accurate bounds for a high-order solution

---

Initialization: Set initial bounds $[lb, ub]$ using the largest and smallest value of the solution f at Lagrange points.

- For each element e of the mesh
  - $depth = 0$
  - Compute extremal control solutions of the element F^e_{min} and F^e_{max}
  - (1) If ($lb > F^e_{min}$ or $ub < F^e_{max}$) and $depth < depth_{max}$
    - Subdivide solution into n sub-solutions.
    - For each sub-solution i
      - Update lb and ub with the value at the vertices for each sub-element.
      - $depth = depth + 1$
      - if ($lb > F^{e_i}_{min}$ or $ub < F^{e_i}_{max}$) re execute the block starting from (1)

Case of edges. Let us have a look on the case of an edge with a solution on it. The solution can be written as follows:

$$f(u) = \sum_{i=0}^{d} B_i^d(u) F_{i(d-i)} \mid \forall u,\ 0 \le u \le 1, \tag{11}$$

where

$$B_i^d(u) = \binom{d}{i} u^i (1-u)^{d-i} \tag{12}$$

are the edge Bernstein polynomials of degree d.

As an example, let us take $d = 2$ (see Fig. 18 on the left). The idea is to apply de Casteljau's algorithm [27] that will subdivide the solution on an edge into two solutions defined on two complementary parts of the edge and whose union is the initial solution.

Let us note f_{ij} the values of f at the Lagrange points, by definition, $f_{02} = F_{02}$ and $f_{20} = F_{20}$. In degree 2, the algorithm is in two steps:

$$F^1_{11} = \frac{F_{20} + F_{11}}{2} \quad F^2_{11} = \frac{F_{11} + F_{02}}{2},$$

$$F^1_{02} = F^2_{20} = \frac{F^1_{20} + F^1_{11}}{2}.$$

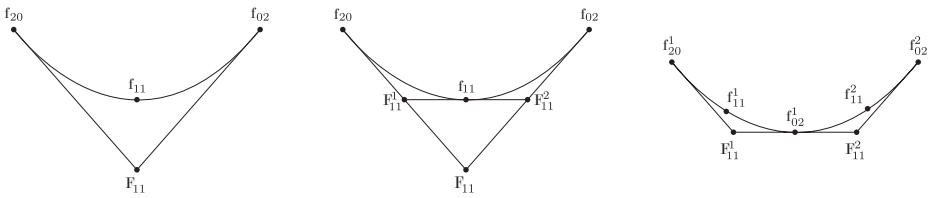


Figure 18: Recursive subdivision of a solution along an edge using a de Casteljau's algorithm

By construction, $F^1_{02} = f_{11}$, and $(f_{20}, F^1_{11}, f_{11})$ (resp. $(f_{11}, F^2_{11}, f_{02})$) are control solutions that define a sub-solution of degree 2 on the first (resp. second) half of the edge. The union of both solutions gives us the initial solution of degree 2 on the whole edge. Note that even if control solution bounds do not still give exact bound, their bounds are closer to the real one after a subdivision step. The principle of this algorithm can be extended as well to other degrees (see [27]).

Case of quadrilaterals. For quadrilaterals, the procedure of subdivision relies on the procedure used for edges. Indeed, the parameterization of a quadrilateral is no more no less the tensor product of the parameterization of two edges and a tensorization of (11) gives:

$$f(u,v) = \sum_{i=0}^{d}\sum_{j=0}^{d} B_i^d(u)B_j^d(v)F_{ij} = \sum_{i=0}^{d} B_i^d(u)(\sum_{j=0}^{d} B_j^d(v)F_{ij}) \mid \forall(u,v),\ 0 \le u,v \le 1, \tag{13}$$

where $B_i^d(u)$ are the edge Bernstein polynomials of degree d given in (12).

Therefore, applying first the previous algorithm for a fixed u (*e.g.* looking all the edges defined by F_{ij} at a fixed i) will generate a set of control solution F^1_{ij} with $0 \le i \le d$ and $0 \le j \le 2d$, By doing the same thing for fixed v will then generate another set F^2_{ij} with $0 \le i, j \le 2d$. By splitting i and j between $\{0, .., d\}$ and $\{d, ..., 2d+1\}$, the set of control solutions can be divided into 4 sets. Each of the set can define a sub-solution of degree d on a quarter of the initial quadrilateral and the union of all the sub-solutions defines a solution of degree d on the initial quadrilateral. More details can be found in [27].

Case of triangles. For triangles, a de Casteljau's algorithm could be applied. However, the set of control solutions generated by the algorithm gives us a subdivision of the solution of the triangle into 3 sub-solutions and this subdivision does not subdivide the edges (whereas it is done with quadrilaterals) which is an issue as our solution may have an extremum on the edges. For this reason, another subdivision process is applied, a process which subdivides our solution into 4 sub-solutions and subdivides it on the edges of the initial triangle as well (see Fig. 19):

The proposed subdivision is based on 3 parameters $0 < \alpha, \beta, \gamma < 1$ (they are usually all equals to $\frac{1}{2}$) that control the subdivision process along each edge. Based on these subdivisions, a subdivision into 4 of the triangle is possible. Let us note a solution f of degree d defined on a triangle:

$$f(u,v,w) = \sum_{i+j+k=d} B^d_{ijk}(u,v,w)F_{ijk}. \tag{14}$$

Each sub-solution f^n with $1 \le n \le 4$ can then be written as:

$$f^n(u,v,w) = \sum_{i+j+k=d} B^d_{ijk}(u,v,w)F^n_{ijk}.$$

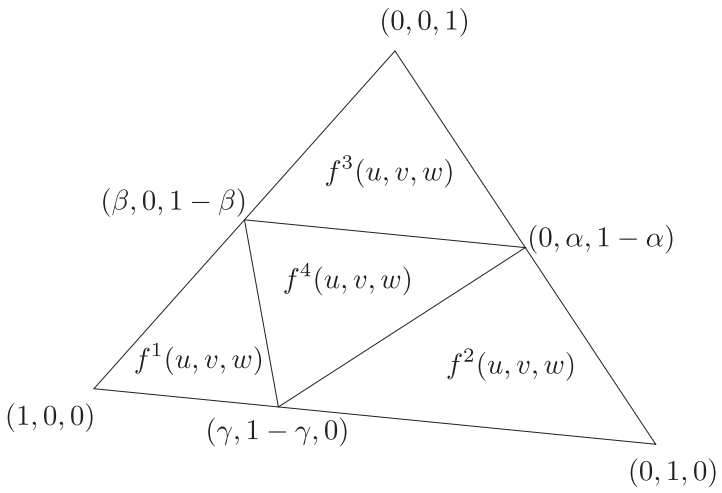


Figure 19: Parts of the triangle where the sub-solutions are created. The points are defined by their parametric coordinates.

with:

$$\begin{cases} F^1_{IJK} = \displaystyle\sum_{j+j_1=J} B^J_{j_1}(\gamma) \sum_{k+k_1=K} B^K_{k_1}(\beta) F_{(I+j_1+k_1)jk}, \\ F^2_{IJK} = \displaystyle\sum_{i+i_1=I} B^I_{i_1}(1-\gamma) \sum_{k+k_1=K} B^K_{k_1}(\alpha) F_{i(i_1+J+k_1)k}, \\ F^3_{IJK} = \displaystyle\sum_{i+i_1=I} B^I_{i_1}(1-\beta) \sum_{j+j_1=J} B^J_{j_1}(1-\alpha) F_{ij(i_1+j_1+K)}, \\ F^4_{IJK} = \displaystyle\sum_{j_1+k_1=I} B^I_{j_1}(\alpha) \sum_{i_1+k_2=J} B^J_{i_1}(\beta) \sum_{i_2+j_2=K} B^K_{i_2}(\gamma) F_{(i_1+i_2)(j_1+j_2)(k_1+k_2)}. \end{cases}$$

The derivation of such control solutions is explained in Appendix. In a same manner as with the quadrilateral and the edge, each of the sub-solution is of degree d on their associated part of the triangle and their union defines the initial solution of degree d on the initial triangle. In practice, the control solutions are first computed along the edges of the element, then along the edges created by the subdivision and then in the inner part of each subdivision.

Thanks to these subdivisions, we are able to compute more accurate bounds for the high-order solution. This is explained in Algorithm 1. By using this recursive algorithm, more accurate bounds for any high-order solution are obtained. Note that all these developments are independent of the degree of the geometrical element where the solution is defined. In Fig. 20, the application of the algorithm changes the initial bounds from [−1 1] (deduced from the values at Lagrange points) to [−1.5032 1.56] which significantly changes the color of the rendered solution.

6.3. Isoline and wire frame rendering

The display of isolines is a common feature used when displaying a numerical (scalar) solution. Usually, classic visualization software use techniques like *marching cubes* to plot isolines. As considered solutions are in the end linear, it just consists in plotting an edge. In our context, these methods do not apply as our solution is not necessarily linear. Our wish is also to have a rendering of isolines which thickness is independent of the considered element and its degree. Also, it should be independent of the zooming level. The proposed solution takes advantage of the built-in functions proposed in the Fragment Shader and uses them to display isolines. The Fragment Shader has the following function `fwidth` which takes in input a scalar variable and outputs an approximate value of the norm of its spatial gradient with respect to the coordinates in the window space. In other words, it gives the variation of your numerical solution on the screen of your computer and it is automatically recomputed at each rendering.

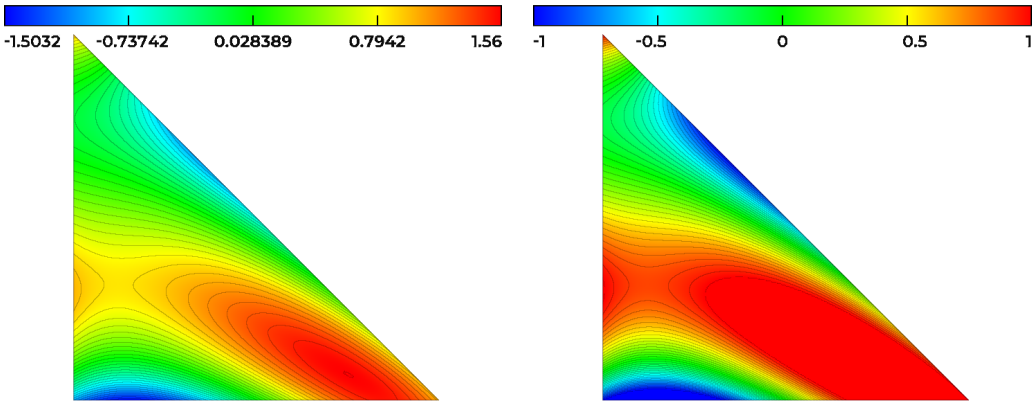


Figure 20: Rendering of P^3-solution of Fig. 4. Left: rendering with a refinement procedure to compute the bounds of the solution for the palette. Right: rendering using the largest and smallest values of the solution at the Lagrange points for the palette. Isolines are also shown.

To understand how to plot an isoline with a wanted thickness, let us have an analysis of the behavior of the numerical solution in the vicinity of an isoline curve [45]. For this purpose, let us consider the numerical solution as a smooth function f, that we will assume at least C^2 and let us define our isoline as

$$I_0(f) = \{x \in \mathbb{R}^d | f(x) = f_0\}. \tag{15}$$

Let us also consider a point $p \in I_0(f)$ such that $\|\nabla f(p)\| \neq 0$. The criterion we use to plot an isoline with a wanted thickness is

$$|f(q) - f_0| < \epsilon \|\nabla f(q)\|, \tag{16}$$

where q is the neighborhood of p and ϵ is a constant thickness. We will show that it controls the size of the isoline. Now we decompose $q - p$ as:

$$q - p = \alpha t + \beta n, \tag{17}$$

where t is a unit tangent vector to $I_0(f)$ and n a unit normal vector to $I_0(f)$ (e.g. aligned with $\nabla f(p)$). This leads to

$$|\nabla f(p)^\intercal (q - p)| = |\beta| . \|\nabla f(p)\|, \tag{18}$$

by definition of the tangent and normal vectors.

A Taylor expansion at the first order in the neighborhood of p with respect to the coordinates of the screen gives us:

$$f(q) = f_0 + \nabla f(p)^\intercal (q - p) + O(\|q - p\|^2). \tag{19}$$

By neglecting terms of order 2, (19) becomes

$$|f(q) - f_0| = |\nabla f(p)^\intercal (q - p)|. \tag{20}$$

As f is C^2, we can also say by continuity that:

$$\nabla f(q) = \nabla f(p) + O(\|q - p\|).$$

By neglecting terms of order 1, we can write that:

$$\nabla f(q) = \nabla f(p). \tag{21}$$

Combining (18), (20) and (21), we get:

$$|f(q) - f_0| = |\beta| . \|\nabla f(p)\|. \tag{22}$$

Finally, if the criterion (16) is true and with (22) and under regularity assumptions ($\|\nabla f(p)\| \neq 0$):

$$|\beta| < \epsilon. \tag{23}$$

Consequently, we notice that Inequality (16) gives a way to control the normal component of the distance between the two points p and q.

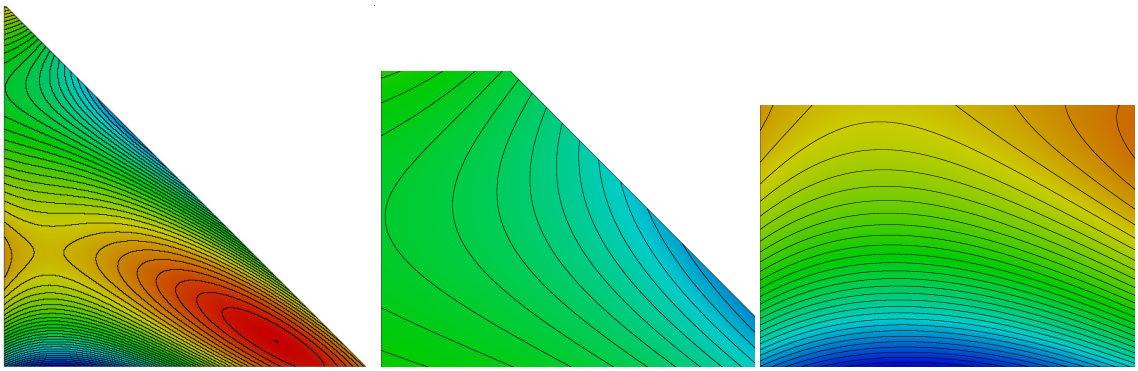


Figure 21: Rendering of P^3-solution of Fig. 4 with its iso values and zoom in various parts.

In practice, for a given f, its two closest iso-values f_0 and f_1 are computed. And for each of these iso-values, Inequality (16) is tested. If it is true, the color associated to the fragment shader is set to the wanted color for the isoline in RGB. The main advantage of this method is that it does not ask any preprocessing to display iso-values. The result of isoline rendering is immediate when asked by the user and gives lines of thickness ϵ in pixels (Fig. 21 shows that the thickness of the isolines remains the same whatever the level of zoom is). Notice that when f is constant or the considered isoline is an extremum or containing a singular point of f ($\|\nabla f(p)\| = 0$), the theory cannot be applied. However, in this case Inequality (16) is always false and therefore no isoline is plotted there.

Using the same approach, a way to properly render the wire frames (e.g. the edges of a geometrical element) is deduced. Indeed, apart from the linear case, where the rendering of such wire frames is straightforward with basic geometry [46], it becomes harder to perform a proper rendering of wire frame with non linear elements. Several problems apply in this case: the mapping between the parameters and the physical coordinates is not linear, but worst of it, the displayed geometry is piece-wise linear, as it is a tessellated geometry. The idea is therefore to apply the previous technique by considering each of the parametric coordinates as a numerical solution. Indeed, a parametric coordinate is defined between 0 and 1 and reaches these values on the boundary of the element. Plotting the isolines associated to 0 and 1 gives thus a way to plot the extremities of any element of any degree with a constant and zoom-independent thickness (Fig. 22). This way anti-aliasing is automatically featured. We also avoid an extra cost which is caused by the classic rendering techniques which plots a set of straight edges on the boundary of the elements. This can take a while when the mesh is of large size where the proposed approach gives an instant result.

7. Numerical examples

7.1. High-order surface meshing

This example is taken from [47] and represents a high-order adapted surface generated from a CAD model. The considered model is a shuttle geometry based on two NURBS of degree 3 defined by 8 (resp. 13) control points and 12 (resp. 17) knots. In Fig. 23, the P^1, P^2 and P^3 meshes are depicted on the left and the errors to the geometry are reported in the right and are represented by their interpolations with P^5, P^8 and P^{10} Lagrange functions.

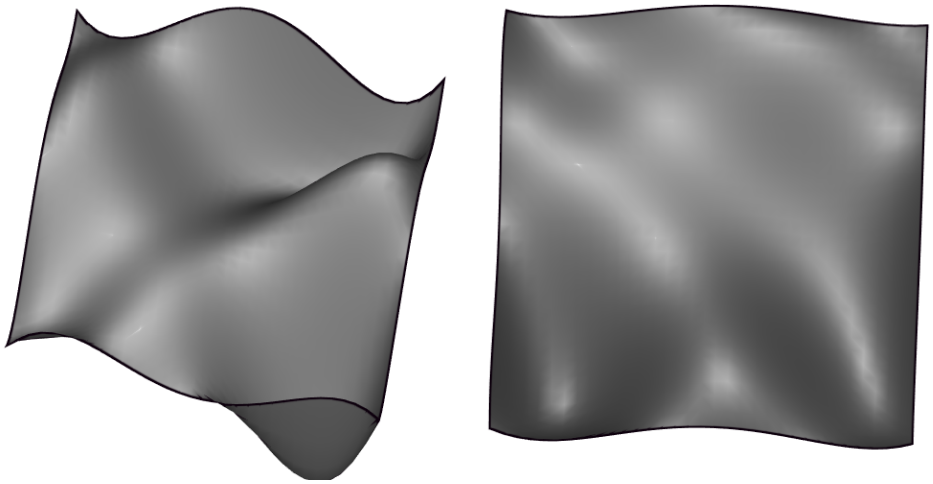


Figure 22: Wire frame rendering performed on a non-planar Q^4-quadrilateral.

7.2. High-order mesh adaptation example

This example is taken from [48]. It shows high-order mesh adaptation. The underlying idea is to develop high-order error estimates so that it minimizes the error induced by the high-order differential of a given numerical solution. The adaptation is performed with respect to a multiscale function. This function oscillates at high frequency and contains small details, it is given by :

$$f_r(x, y, z) = 8\, xyz\, \sin(5\,\pi\, xyz)^4 + \frac{1}{10}\left(1 - (\sin(5\,\pi\, xyz)^4\right)^8\, \cos(100\,\pi\, xyz).$$

The high frequencies variations of the function are depicted in Figs. 24 and 25. The final mesh for P^1 contains 2 559 432 vertices, 564 774 triangles and 14 621 126 tetrahedra while the equivalent mesh for P^4 in terms of degrees of freedom contains 39 612 vertices, 19 710 triangles and 208 197 tetrahedra.

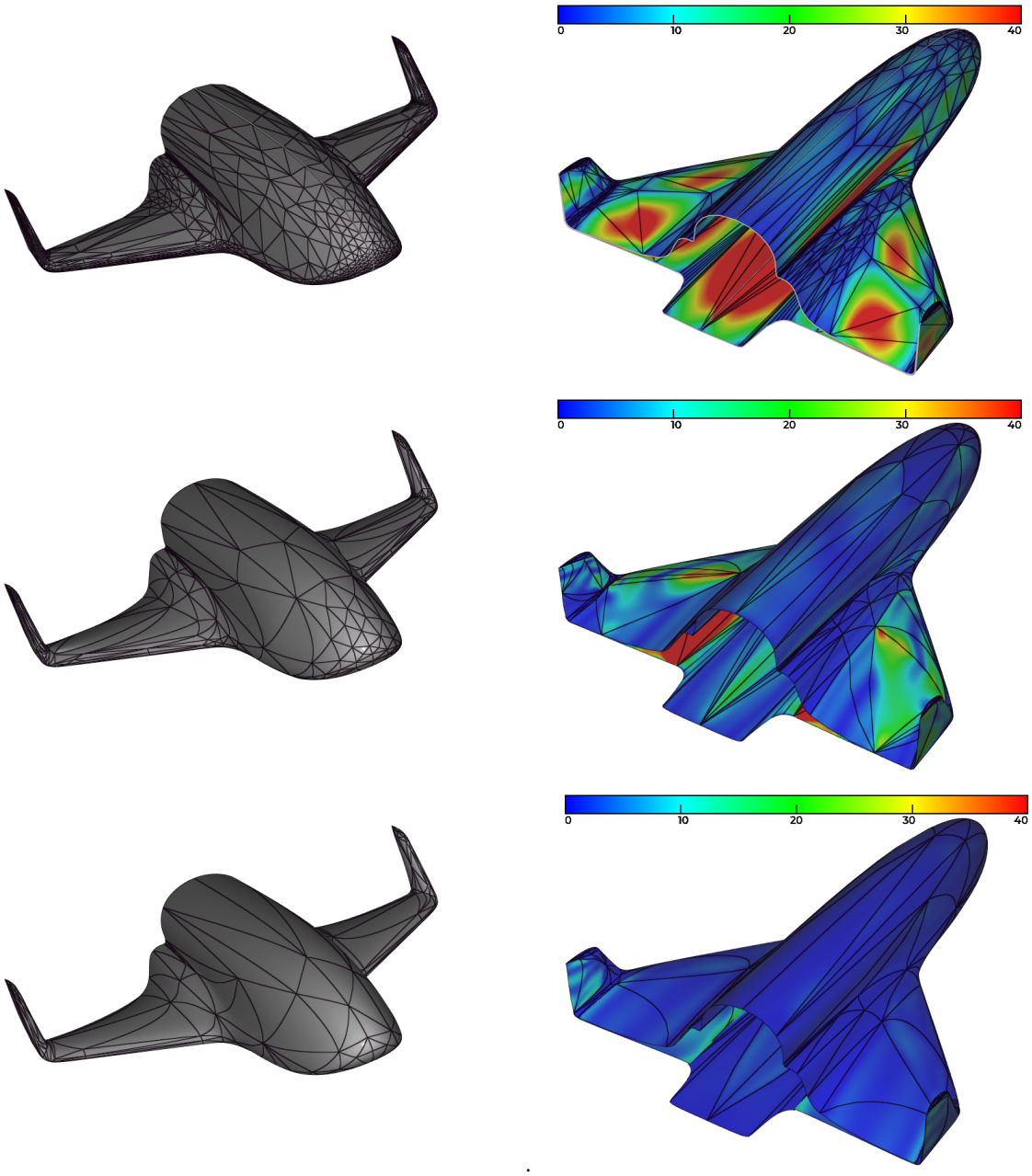
In all the figures, we assess the interest of mesh adaptation for high-order functions. In particular, we show no loss of accuracy in the solution rendering while the mesh (and the number of used degrees of freedom) is coarsened and the order of the solution increased. Note that, as all the meshes are linear, the rendering is pixel-exact.

7.3. High-order boundary element solution

This example is taken from [49]. It shows a P^3-geometry of an unarmed F15 aircraft. The considered problem is the scattering of plane waves using adaptive Boundary Element Method (BEM). In Fig. 26, the resolution of a P^3 BEM on the aircraft is shown. In Fig. 27, the used P^3-mesh is shown with various tessellation levels. We clearly observe the diffraction phenomenon in a high-order fashion as it takes into account the high-order features of the geometry.

7.4. Wave propagation with Perfectly Matched Layers (PML)

This example is based on the numerical simulation explained in [50]. It is a wave propagation inside the cube $[-1,\ 1]^3$ with a PML boundary condition on the left ($y = -1$) and on the right ($y = 1$), and a Dirichlet boundary condition at the top ($z = 1$) and at the bottom ($z = -1$). It is solved on a structured mesh of 8000 hexahedra with a Q^6 solution using Gauss-Lobatto quadrature points. The resolution is done with spectral finite elements and a Leap-Frog scheme for time integration. In Figs. 28 and 29, we show the mesh and the solution at the following time step on the volume elements lying along $x = 0$, $y = 0$ and $z = 0$. We clearly observe the absorbing boundary condition at the boundary $x = 1$ and the reflecting waves at the boundary $y = 1$. For this case, the hexahedral rendering is pixel-exact, as the geometrical elements are truly linear.



.

Figure 23: P^1 (top), P^2 (middle) and P^3 (bottom) meshes and point-wise distance to the shuttle geometry.

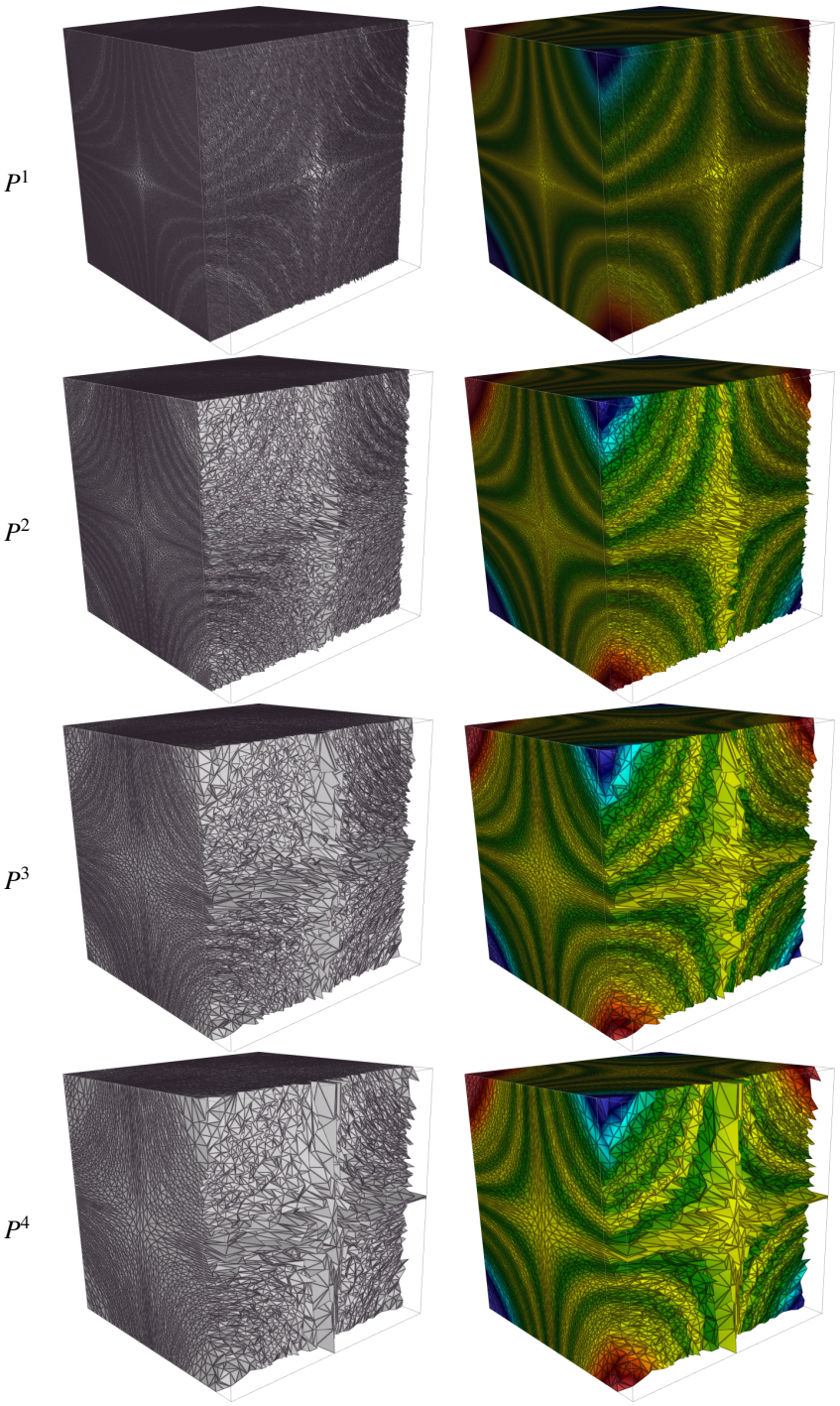


Figure 24: From top to bottom, P^1 to P^4 adapted meshes (left) with the corresponding solution on it (right).

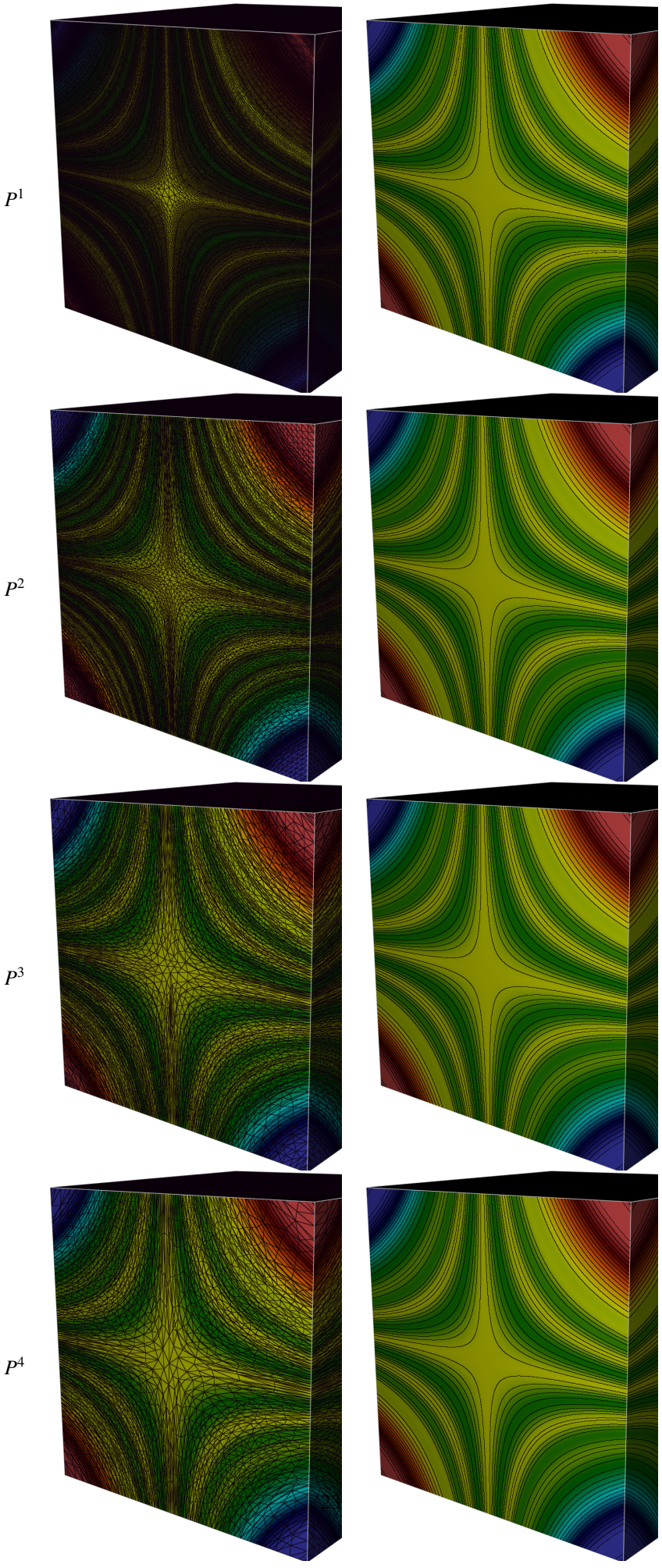


Figure 25: From top to bottom, P^1 to P^4 adapted meshes with the solution (left) and iso values (right).

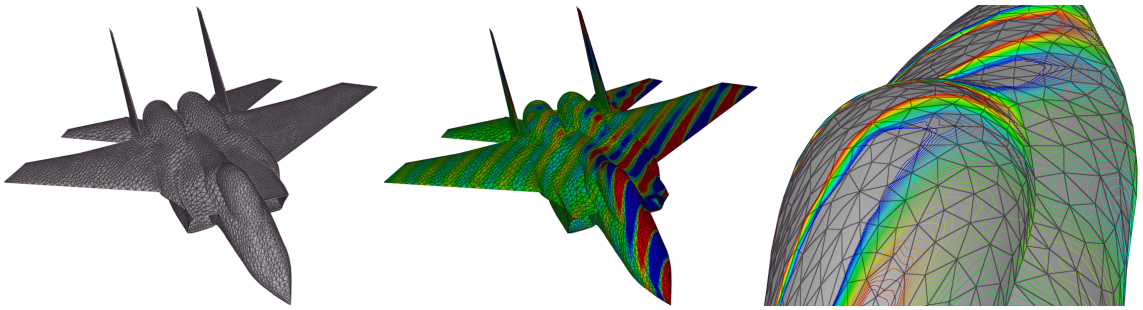


Figure 26: High-order BEM resolution of the scattering of plane waves on an aircraft. Left: mesh. Middle: mesh with solution. Right: zoom on isolines around the cockpit. Courtesy of Stéphanie Chaillat (ENSTA).

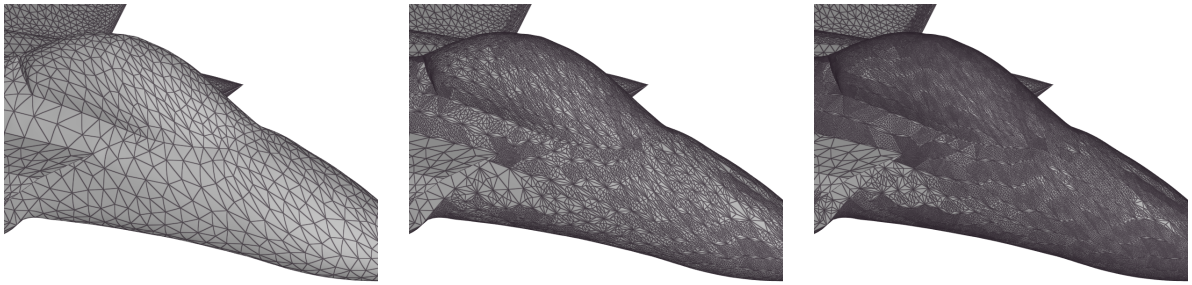


Figure 27: Left, zoom on a curved part of the P^3-mesh used for the BEM resolution. Middle and right, various tessellation levels used for the display of the geometry. Courtesy of Stéphanie Chaillat (ENSTA).

7.5. Computational AeroAcoustics

This example is from [51]. It is based on the resolution of two equations. First, Euler equations solved with Discontinuous Galerkin Method provides the mean flow. Then, linearized Euler equations solved with Discontinuous Galerkin Method provides the acoustic field. In Fig. 30, the method is shown on an engine containing a central body. The domain of computation is bounded with the 4 boundaries: inlet, wall, outlet and wall. The mean flow is computed on Q^2 mesh with Q^{10} solution (top) and the propagation of a small perturbation is computed over the mean flow using a Q^8 solution (bottom). We clearly observe the variations of the mean flow as well as the small perturbations induces by it.

7.6. Waveguide solution

This example is taken from [52]. This shows a procedure for the generation of accurate polychromatic sources in waveguides which takes into account mode dispersion. This method is implemented using a Discontinuous Galerkin Time-Domain (DGTD) solver. In Fig. 31, E_x field during the conversion of a polychromatic TE_{01} mode to a TE_{02} mode is shown, with a zoom in Fig. 32. The displayed mesh is P^3 and the solution is P^3 as well.

7.7. Performance

In this section, we sum up the CPU times needed for all previous examples to open the mesh and solutions files and the total time to get the first image (i.e. total time to initialize the window, open files and add all the objects to the scene to render. These times are sum up in Table 1. The memory used (in Mb) and the number of Frames Per Seconds (FPS) are also added in this table. All the results collected in Table 1 have been generated with the same laptop: a MacBook Pro with 2.6 GHz 6-core Intel Core i7 with 32 Gb of RAM, and the GPU is a AMD Radeon Pro Vega 20 4 Gb. We use the `libMeshb` library [53] to efficiently read/write mesh/solution files.

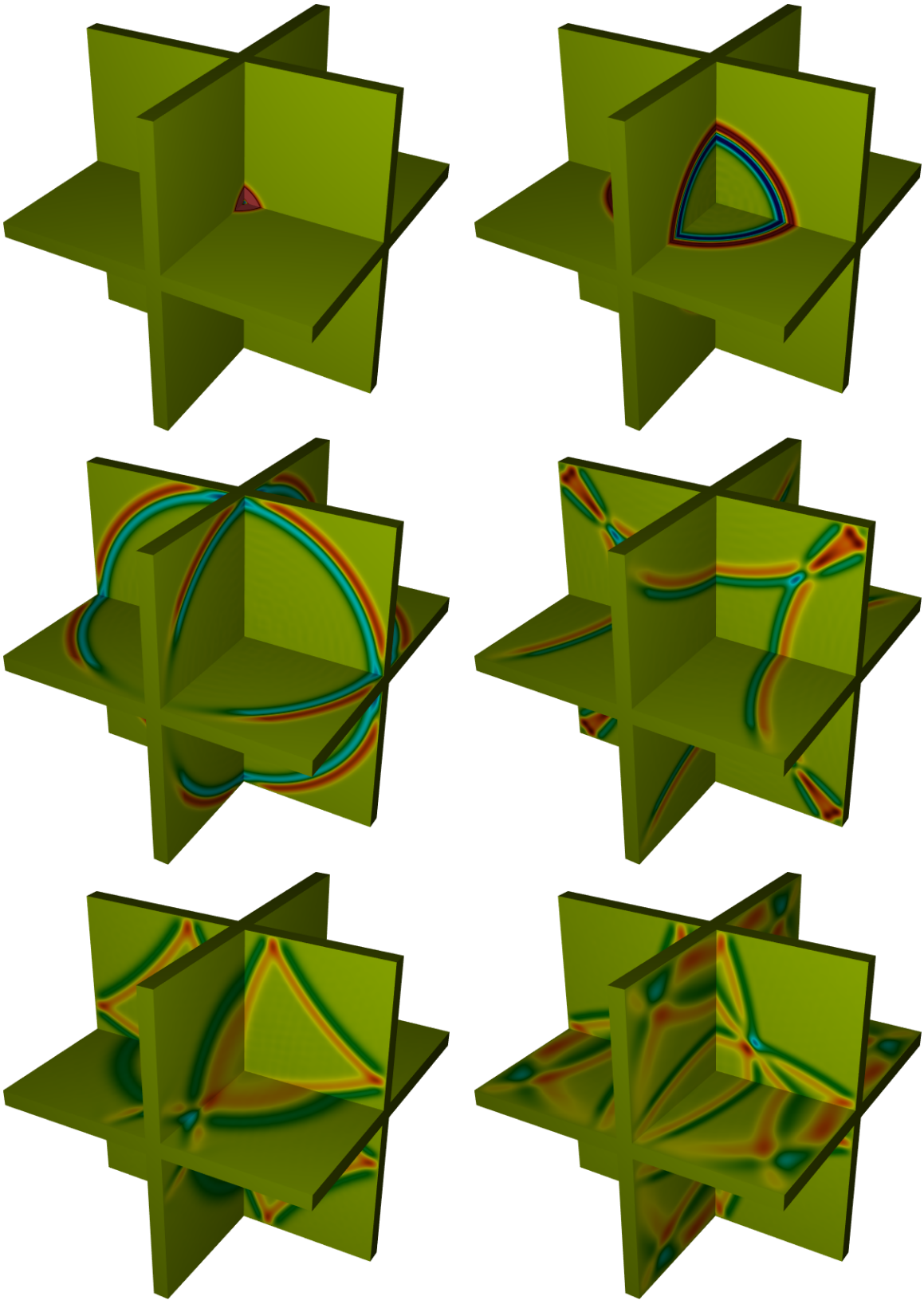


Figure 28: Q^6 solution of a wave propagation problem on an hexahedral mesh at different times. Courtesy of Sébastien Impériale (Inria).

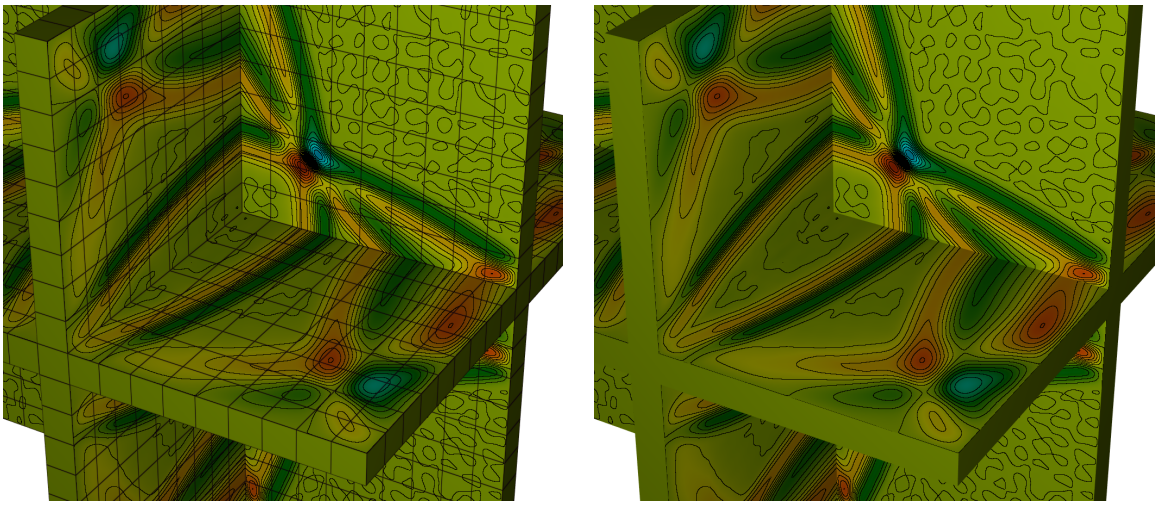


Figure 29: Zoom in the middle of the cube with the solution of Fig. 28 with its isolines. Left: the structured mesh of hexahedra is also shown. Courtesy of Sébastien Impériale (Inria).

Case	# vertices	Mesh / Sol. degrees	Mesh / Sol. opening (s)	Total time (s)	Memory (Mb)	FPS
P^1-shuttle	1 647	1 / 5	0.01 / 0.02	0.43	57.3	60
P^2-shuttle	1 690	2 / 8	0.01 / 0.02	0.65	50.1	60
P^3-shuttle	1 791	3 / 10	0.01 / 0.01	0.91	60.3	60
P^1-cube	2 559 432	1 / 1	0.39 / 0.71	1.57	954.0	60
P^2-cube	307 368	1 / 2	0.05 / 0.22	0.58	254.3	60
P^3-cube	78 409	1 / 3	0.02 / 0.12	0.44	139.8	60
P^4-cube	39 612	1 / 4	0.01 / 0.11	0.41	115.9	60
F15	250 022	3 / 3	0.01 / 0.01	0.36	64.6	60
PML	9 261	1 / 6	0.01 / 0.04	0.52	96.3	60
NACA 1	1 404	2 / 10	0.01 / 0.03	1.88	73.6	60
NACA 2	1 404	2 / 8	0.01 / 0.02	0.92	61.9	60
Wave-guide	518 200	3 / 3	1.03 / 0.74	2.10	125.8	60

Table 1: CPU times (in seconds), memory used (in Mb) and number of Frames Per Seconds (FPS) to render the numerical solutions of this section

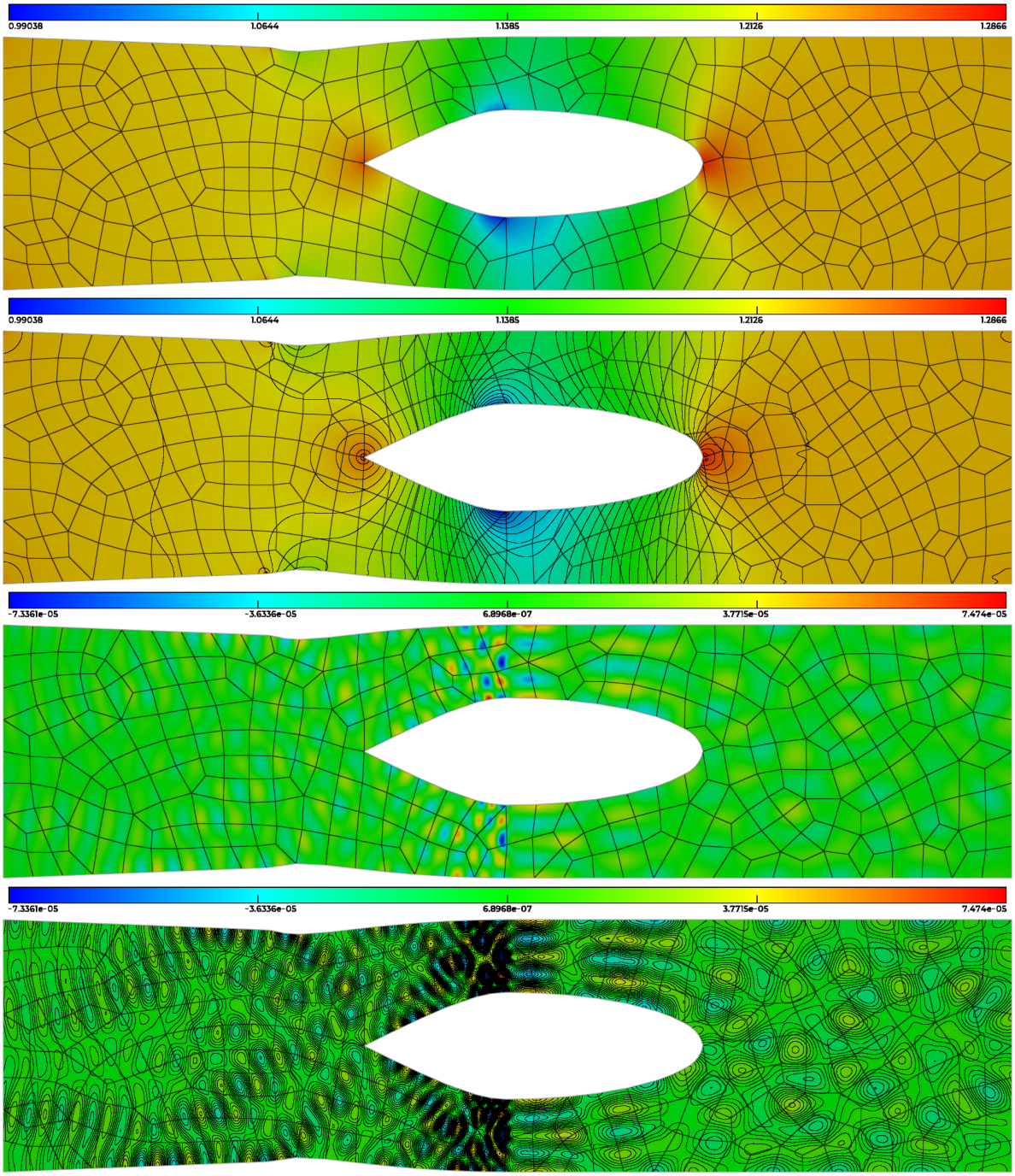


Figure 30: Top (two first pictures), mean flow around an engine using a Q^2 mesh with Q^{10} solution. Bottom (two last pictures), perturbation over the mean flow using a Q^8 solution. Courtesy of Christophe Peyret (ONERA).

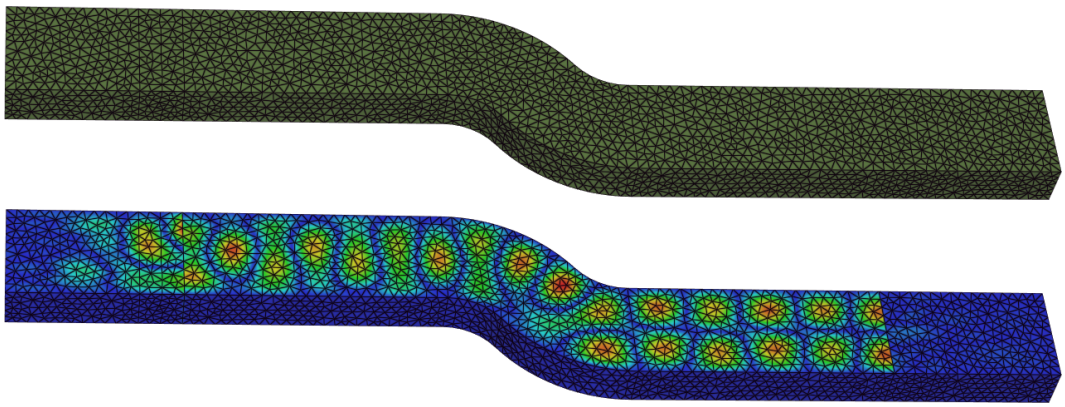


Figure 31: From top to bottom, meshes and solutions of a conversion of a polychromatic TE_{01} mode to a TE_{02} mode. Courtesy of Alexis Gobé and Jonathan Viquerat (Inria).

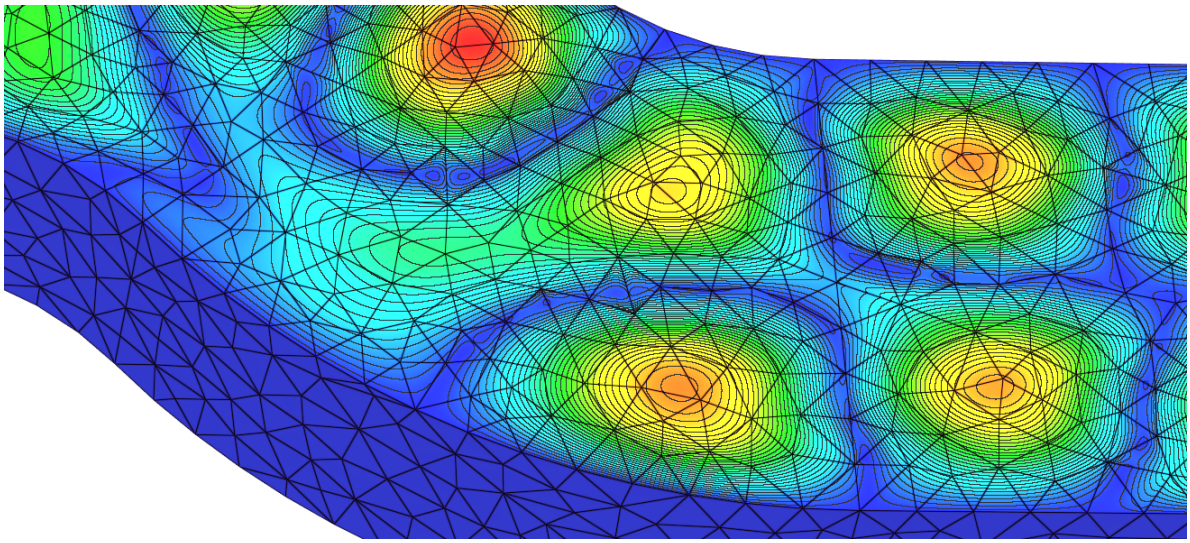


Figure 32: Zoom around the curved part of the mesh with the waveguide solution of Fig. 31. Courtesy of Alexis Gobé and Jonathan Viquerat (Inria).

8. Conclusion

A novel approach to high-order mesh and solution visualization has been presented. This method is based on OpenGL 4 framework and enables a full GPU-based rendering of high-order entities and a reliable rendering of high-order solutions on it. The latter is even pixel exact when it is defined on linear elements. The OpenGL framework and its use in our case has been exposed. Geometrical properties of the high-order elements have been discussed in order to optimize their discretization by the GPU. Also, numerical properties of the high-order solutions have been studied in order to get appropriate bounds for the solution and therefore set a pertinent palette for the user. A method to plot isolines and perform wire frame rendering on any kind of elements has been developed. This method is based on implicit curves rasterization methods and does not need a marching square method. Finally several examples illustrate the ability of the technology to handle high-order meshes and solutions in various applications. All these developments are part of a the new generation of `Vizir` [24] software, `Vizir` 4 which can be downloaded in its dedicated web site `https://vizir.inria.fr`, that is developed at Inria by GAMMA team. The software is at the time able to render any geometry from degree 1 to 4 and any polynomial solution from degree 0 to 10. There is no technical issue to increase the maximal degree. The only requirement is to add additional shape function evaluation in the fragment shader. However, at the time, most of meshes are at most degree three and it is not common to deal with very high-order solutions.

Perspectives for this work are about the volume rendering. Indeed, further developments to perform the capping (e.g the computation of the exact intersection of volume mesh with a cut plane) and in particular with elements of order greater than one. Also, some developments are needed for the proper rendering of iso-surfaces. For the latter, classic rendering might be used. Finally, CAD rendering could also be considered.

Acknowledgments

This work was supported by the public grants : *Investissement d'avenir* project, reference ANR-11-LABX-0056-LMH, LabEx LMH and ANR Impacts, reference ANR-18-CE46-0003.

The authors are also grateful to Sébastien Impériale, Alexis Gobé, Loïc Maréchal (Inria) and Christophe Peyret (ONERA) for giving them some high-order meshes and solutions and to Patrick Laug, Paul Louis George and Julien Vanharen (Inria) for testing the software.

Appendix: Subdivision of a solution defined on a triangle into 4 sub-solutions

The idea is to perform an exact subdivision of a solution of degree d defined on a triangle into 4 thanks to the three points defined via their parametric coordinates given in Fig. 19 with $0 < \beta, \gamma, \alpha < 1$ and to express the control solutions of these sub-triangles. As explained in Fig. 19, a new parameterization for each sub-triangle can be deduced.

Let us have a look at the sub-solution on the triangle defined defined by the triplet $\{(1,0,0);(\gamma,1-\gamma,0);(\beta,0,1-\beta)\}$. It can be parameterized with new parametric coordinates $0 \leq u, v, w \leq 1$ as follows:

$$f^1(u,v,w) = f(u+\beta w+\gamma v,(1-\gamma)v,(1-\beta)w).$$

Let us expand this formula to obtain an expression of the Bézier parameterization of this sub-solution. Following the definition of f given in (14):

$$f^1(u,v,w) = \sum_{i+j+k=d} B^d_{ijk}(u+\beta w+\gamma v,(1-\gamma)v,(1-\beta)w)F_{ijk}. \tag{24}$$

By definition of Bernstein polynomials, (24) becomes:

$$f^1(u,v,w) = \sum_{i+j+k=d} F_{ijk}\frac{d!}{i!j!k!}(u+\beta w+\gamma v)^i((1-\gamma)v)^j((1-\beta)w)^k, \tag{25}$$

If we notice that :

$$(u+\beta w+\gamma v)^i = \sum_{i_1+j_1+k_1=i} \frac{i!}{i_1!j_1!k_1!}u^{i_1}\beta^{k_1}w^{k_1}\gamma^{j_1}v^{j_1}. \tag{26}$$

then we obtain from (25) and (26):

$$f^1(u,v,w) = \sum_{i+j+k=d} F_{ijk} \frac{d!}{i!j!k!} (1-\gamma)^j v^j (1-\beta)^k w^k \left(\sum_{i_1+j_1+k_1=i} \frac{i!}{i_1!j_1!k_1!} u^{i_1} \beta^{k_1} w^{k_1} \gamma^{j_1} v^{j_1} \right)$$

Now, we concatenate all the combinations that define the two sums in one sum:

$$\begin{aligned}
f^1(u,v,w) &= \sum_{i_1+j_1+k_1+j+k=d} \frac{d!}{i_1!j_1!k_1!j!k!} F_{(i_1+j_1+k_1)jk} (1-\gamma)^j v^j (1-\beta)^k w^k u^{i_1} \beta^{k_1} w^{k_1} \gamma^{j_1} v^{j_1} \\
&= \sum_{i_1+j_1+k_1+j+k=d} \left(\frac{d!}{i_1!(k+k_1)!(j+j_1)!} u^{i_1} v^{j+j_1} w^{k+k_1} \right) \\
&\qquad \left(\frac{(k+k_1)!(j+j_1)!}{k!k_1!j!j_1!} \beta^{k_1} \gamma^{j_1} (1-\gamma)^j (1-\beta)^k \right) F_{(i_1+j_1+k_1)jk} \\
&= \sum_{i_1+J+K=d} B^d_{i_1 JK}(u,v,w) \sum_{\substack{k+k_1=K \\ j+j_1=J}} \left(\frac{K!J!}{k!k_1!j!j_1!} \beta^{k_1} \gamma^{j_1} (1-\gamma)^j (1-\beta)^k \right) F_{(i_1+j_1+k_1)jk} \\
&= \sum_{I+J+K=d} B^d_{IJK}(u,v,w) F^1_{IJK}.
\end{aligned}$$

The control solutions of the sub-solution can be therefore expressed using the Bernstein polynomials as:

$$\boxed{F^1_{IJK} = \sum_{j+j_1=J} B^J_{j_1}(\gamma) \sum_{k+k_1=K} B^K_{k_1}(\beta) F_{(I+j_1+k_1)jk}.}$$

Using same derivations on sub-solutions defined on triangles $\{(0,0,1);(0,\alpha,1-\alpha);(\gamma,1-\gamma,0)\}$ and $\{(0,0,1);(\beta,0,1-\beta);(0,\alpha,1-\alpha)\}$, we have:

$$\begin{aligned}
f^2(u,v,w) &= f(\gamma u, v + (1-\gamma)u + \alpha w, (1-\beta)w) = \sum_{I+J+K=d} B^d_{IJK}(u,v,w) F^2_{IJK}, \\
f^3(u,v,w) &= f(\beta u, \alpha v, w + (1-\beta)u + (1-\alpha)v) = \sum_{I+J+K=d} B^d_{IJK}(u,v,w) F^3_{IJK}.
\end{aligned}$$

and the associated control solutions are:

$$\boxed{F^2_{IJK} = \sum_{i+i_1=I} B^I_{i_1}(1-\gamma) \sum_{k+k_1=K} B^K_{k_1}(\alpha) F_{i(i_1+J+k_1)k},}$$

$$\boxed{F^3_{IJK} = \sum_{i+i_1=I} B^I_{i_1}(1-\beta) \sum_{j+j_1=J} B^J_{j_1}(1-\alpha) F_{ij(i_1+j_1+K)}.}$$

Finally, for the sub-triangle $\{(\gamma,1-\gamma,0);(\beta,0,1-\beta);(0,\alpha,1-\alpha)\}$, its parameterization is:

$$f^4(u,v,w) = f(\beta v + \gamma w, \alpha u + (1-\gamma)w, (1-\alpha)u + (1-\beta)v).$$

Let us expand this formula to obtain the control solutions:

$$
\begin{aligned}
f^4(u,v,w) &= \sum_{i+j+k=d} F_{ijk} B^d_{ijk}(\beta v + \gamma w, \alpha u + (1-\gamma)w, (1-\alpha)u + (1-\beta)v) \\
&= \sum_{i+j+k=d} F_{ijk} \frac{d!}{i!j!k!} (\beta v + \gamma w)^i (\alpha u + (1-\gamma)w)^j ((1-\alpha)u + (1-\beta)v)^k \\
&= \sum_{i+j+k=d} F_{ijk} \frac{d!}{i!j!k!} \left\{ \sum_{i_1+i_2=i} \frac{i!}{i_1!i_2!} \beta^{i_1} \gamma^{i_2} v^{i_1} w^{i_2} \right\} \left\{ \sum_{j_1+j_2=j} \frac{j!}{j_1!j_2!} \alpha^{j_1} (1-\gamma)^{j_2} u^{j_1} w^{j_2} \right\} \\
&\qquad \left\{ \sum_{k_1+k_2=k} \frac{k!}{k_1!k_2!} (1-\alpha)^{k_1} (1-\beta)^{k_2} u^{k_1} v^{k_2} \right\} \\
&= \sum_{i_1+i_2+j_1+j_2+k_1+k_2=d} F_{(i_1+i_2)(j_1+j_2)(k_1+k_2)} \frac{d!}{i_1!i_2!j_1!j_2!k_1!k_2!} \\
&\qquad \beta^{i_1} \gamma^{i_2} \alpha^{j_1} (1-\gamma)^{j_2} (1-\alpha)^{k_1} (1-\beta)^{k_2} u^{j_1+k_1} v^{i_2+k_2} w^{i_2+j_2} \\
&= \sum_{I+J+K=d} B^d_{IJK}(u,v,w) \\
&\left(\sum_{\substack{j_1+k_1=I \\ i_1+k_2=J \\ i_2+j_2=K}} \frac{I!J!K!}{i_1!i_2!j_1!j_2!k_1k_2!} \beta^{i_1} \gamma^{i_2} \alpha^{j_1} (1-\gamma)^{j_2} (1-\alpha)^{k_1} (1-\beta)^{k_2} F_{(i_1+i_2)(j_1+j_2)(k_1+k_2)} \right) \\
= \sum_{I+J+K=d} & B^d_{IJK}(u,v,w) F^4_{IJK}.
\end{aligned}
$$

The control solutions of the sub-solution can be therefore expressed using the Bernstein polynomials:

$$
\boxed{F^4_{IJK} = \sum_{j_1+k_1=I} B^I_{j_1}(\alpha) \sum_{i_1+k_2=J} B^J_{i_1}(\beta) \sum_{i_2+j_2=K} B^K_{i_2}(\gamma) F_{(i_1+i_2)(j_1+j_2)(k_1+k_2)}.}
$$

References

[1] L. Piegl, W. Tiller, The NURBS book, Springer, 1995.

[2] H. Borouchaki, P. Laug, P. L. George, Parametric surface meshing using a combined advancing-front – generalized-Delaunay approach, International Journal for Numerical Methods in Engineering (2000) 233–259.

[3] R. Aubry, S. Dey, E. L. Mestreau, B. K. Karamete, D. Gayman, A robust conforming NURBS tessellation for industrial applications based on a mesh generation approach, Computer-Aided Design (2015) 3–42.

[4] P. J. Frey, P. L. George, Mesh Generation: application to finite elements, 2nd Edition, John Wiley & Sons, 2008.

[5] P. L. George, H. Borouchaki, E. Saltel, Ultimate robustness in meshing an arbitrary polyhedron, International Journal for Numerical Methods in Engineering 58 (7) (2003) 1061–1089.

[6] L. Maréchal, Advances in octree-based all-hexahedral mesh generation: handling sharp features, Proceedings of the 18th international meshing roundtable (2009) 65–84.

[7] A. Loseille, Unstructured Mesh Generation and Adaptation, in: R. Abgrall, C.-W. Shu (Eds.), Handbook of Numerical Methods for Hyperbolic Problems, Vol. 18 of Handbook of Numerical Analysis, Elsevier, 2017, Ch. 10, pp. 263 – 302.

[8] G. E. Forsythe, W. R. Wasow, Finite-difference methods for partial differential equations, Wiley, 1960.

[9] P. G. Ciarlet, The Finite Element Method for Elliptic Problems, North-Holland Publishing Company, 1978.

[10] B. Van Leer, Towards the ultimate conservative difference scheme. A second-order sequel to Godunov's method, Journal of Computational Physics (1979) 101–136.

[11] J. S. Hesthaven, T. Warburton, Nodal Discontinuous Galerkin Methods: algorithms, analysis and applications, Springer Publishing Company, Incorporated, 2008.

[12] F. Bassi, S. Rebay, High-order accurate discontinuous finite element solution of the 2D Euler equations, Journal of computational physics 138 (2) (1997) 251–285.

[13] F. Haider, P. Brenner, B. Courbet, J.-P. Croisille, Parallel implementation of k-exact Finite Volume reconstruction on unstructured grids, in: High order nonlinear numerical schemes for evolutionary PDEs, Springer International Publishing, 2014, pp. 59 – 75.

[14] J. Vanharen, G. Puigt, X. Vasseur, J.-F. Boussuge, P. Sagaut, Revisiting the spectral analysis for high-order spectral discontinuous methods, Journal of Computational Physics 337 (2017) 379 – 402.
[15] T. J. R. Hughes, J. A. Cottrell, Y. Bazilevs, Isogeometric analysis: CAD, finite elements, NURBS, exact geometry and mesh refinement, Computer methods in applied mechanics and engineering (2005) 4135–4195.
[16] S. Dey, R. M. O'bara, M. S. Shephard, Curvilinear Mesh Generation in 3D, in: Proceedings of the 7th International Meshing Roundtable, 1999, pp. 407–417.
[17] S. J. Sherwin, J. Peiró, Mesh generation in curvilinear domains using high-order elements, International Journal for Numerical Methods in Engineering 53 (1) (2002) 207–223.
[18] P. L. George, H. Borouchaki, Construction of tetrahedral meshes of degree two, International Journal for Numerical Methods in Engineering 90 (9) (2012) 1156,1182.
[19] X. Roca, A. Gargallo-Peiró, J. Sarrate, Defining Quality Measures for High-Order Planar Triangles and Curved Mesh Generation, in: W. R. Quadros (Ed.), Proceedings of the 20th International Meshing Roundtable, Springer Berlin Heidelberg, 2012, pp. 365–383.
[20] A. Johnen, J.-F. Remacle, C. Geuzaine, Geometrical validity of curvilinear finite elements, Journal of Computational Physics 233 (15) (2013) 359–372.
[21] T. Toulorge, C. Geuzaine, J.-F. Remacle, J. Lambrechts, Robust untangling of curvilinear meshes, Journal of Computational Physics 254 (2013) 8 – 26.
[22] R. Feuillet, A. Loseille, F. Alauzet, P2 mesh optimization operators, in: X. Roca, A. Loseille (Eds.), 27th International Meshing Roundtable, Springer International Publishing, Cham, 2018, pp. 3–21.
[23] P. Zwanenburg, S. Nadarajah, On the necessity of superparametric geometry representation for discontinuous galerkin methods on domains with curved boundaries, in: 23rd AIAA Computational Fluid Dynamics Conference, 2017, p. 3946.
[24] A. Loseille, R. Feuillet, Vizir: High-order mesh and solution visualization using OpenGL 4.0 graphic pipeline, in: 2018 AIAA Aerospace Sciences Meeting, 2018, pp. AIAA 2018–1174.
[25] D. Wolff, OpenGL 4.0 Shading Language Cookbook, Packt Publishing, 2011.
[26] G. Sellers, R. Wright, N. Haemel, OpenGL SuperBible, Sixth Edition, Addison-Wiley, 2013.
[27] H. Borouchaki, P. L. George, Meshing, Geometric Modeling and Numerical Simulation 1: Form Functions, Triangulations and Geometric Modeling, John Wiley & Sons, 2017.
[28] R. Feuillet, Embedded and high-order meshes: two alternatives to linear body-fitted meshes, Ph.D. thesis, Paris Saclay (2019).
[29] KitWare Inc., ParaView, `https://www.paraview.org/`.
[30] TecPlot Inc., TecPlot, `https://www.tecplot.com/`.
[31] Intelligent Light, FieldView, `http://www.ilight.com/en/products/fieldview-18`.
[32] Ansys Inc., Ensight, `https://www.ansys.com/en/products/platform/ansys-ensight`.
[33] P. J. Frey, Medit: An interactive mesh visualization software, INRIA Technical Report RT0253 (2001).
[34] A. Loseille, H. Guillard, A. Loyer, An introduction to Vizir: an interactive mesh visualization and modification software, EOCOE, Rome, Italy (2016).
[35] C. Geuzaine, J.-F. Remacle, Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities, International Journal for Numerical Methods in Engineering 79 (11) (2009) 1309–1331.
[36] A. Vlachos, P. Jörg, C. Boyd, J. L. Mitchell, Curved PN Triangles, in: Proceedings of the 2001 Symposium on Interactive 3D Graphics, 2001, pp. 159–166.
[37] J.-F. Remacle, N. Chevaugeon, E. Marchandise, C. Geuzaine, Efficient visualization of high-order finite elements, International Journal for Numerical Methods in Engineering 69 (5) (2007) 750–771.
[38] M. Maunoury, C. Besse, V. Mouysset, S. Pernet, P.-A. Haas, Well-suited and adaptive post-processing for the visualization of hp simulation results, Journal of Computational Physics 375 (2018) 1179 – 1204.
[39] M. Maunoury, Méthode de visualisation adaptée aux simulations d'ordre élevé. Application à la compression-reconstruction de champus rayonnés pour des ondes harmoniques, Thèse de l'Université de Toulouse (2019).
[40] M. Rasquin, A. Bauer, K. Hillewaert, Scientific post hoc and in situ visualisation of high-order polynomial solutions from massively parallel simulations, International Journal of Computational Fluid Dynamics 33 (4) (2019) 171–180. `doi:10.1080/10618562.2019.1618453`.
[41] B. Nelson, R. Haimes, R. M. Kirby, GPU-Based Interactive Cut-Surface Extraction From High-Order Finite Element Fields, IEEE Transactions on Visualization and Computer Graphics 17 (12) (2011) 1803–11.
[42] B. Nelson, E. Liu, R. M. Kirby, R. Haimes, ElVis: A System for the Accurate and Interactive Visualization of High-Order Finite Element Solutions, IEEE Transactions on Visualization and Computer Graphics 18 (12) (2012) 2325–2334.
[43] J. Peiró, D. Moxey, B. Jordi, S. J. Sherwin, B. Nelson, R. M. Kirby, R. Haimes, High-Order Visualization with ElVis, in: Notes on Numerical Fluid Mechanics and Multidisciplinary Design, Springer International Publishing, 2015, pp. 521–534.
[44] P. L. George, H. Borouchaki, F. Alauzet, P. Laug, A. Loseille, L. Maréchal, Meshing, Geometric Modeling and Numerical Simulation 2: Metrics, Meshes and Mesh Adaptation, John Wiley & Sons, 2019.
[45] G. Taubin, Distance approximations for rasterizing implicit curves, in: ACM Trans. Graph., Vol. 13, 1994, pp. 3–42.
[46] NVIDIA, Whitepaper WP-03014-001_v01.
[47] R. Feuillet, O. Coulaud, A. Loseille, Anisotropic Error Estimate for High-Order Parametric Surface Mesh Generation, in: 28th International Meshing Roundtable, 2019, pp. 1 – 15.
[48] O. Coulaud, A. Loseille, Very High Order Anisotropic Metric-Based Mesh Adaptation in 3D, Procedia Engineering 163 (2016) 353–364.
[49] S. Chaillat, S. Groth, A. Loseille, Metric-based anisotropic mesh adaptation for 3D acoustic boundary element methods , Journal of Computational Physics 372 (2018) 473–499.
[50] D. H. Baffet, M. J. Grote, S. Impériale, M. Kachanovska, Energy Decay and Stability of a Perfectly Matched Layer For the Wave Equation, Journal of Scientific Computing 81 (3) (2019) 2237–2270.
[51] C. Peyret, A Full High Order Method for Computational AeroAcoustics, in: 23rd AIAA/CEAS Aeroacoustics Conference, 2017.
[52] J. Viquerat, Efficient time-domain numerical analysis of waveguides with tailored wideband pulses, Microwave and Optical Technology

Letters 61 (6) (2019) 1534–1539.
[53] L. Maréchal, The libMeshb library: An easy way to access files in Gamma Mesh Format, https://github.com/LoicMarechal/libMeshb, [Online; accessed 31-August-2020].